International Journal of Computer Science Engineering Techniques – Volume 6 Issue 2, April - 2022

Real-Time Fraud Detection and Feature Store Design Patterns for Streaming ML in Financial Services

Jeevan Krishna Paruchuri

Independent Researcher
paruchuri.g167@gmail.com

Abstract—Real-time fraud detection in payment authorization workflows imposes a particularly demanding combination of constraints. Tens of thousands of transactions per second must be scored within an end-to-end latency budget of two hundred milliseconds. The features that the scoring model consumes must reflect a recent enough view of the underlying entities to capture fraudulent activity that occurred seconds earlier. The system must continue to detect fraud even under partial component failure. This paper presents a case study of a production fraud detection system operated at a large payments processor. It processes between ten thousand and fifty thousand transactions per second under a two-hundred-millisecond SLA. The system combines event-driven ingestion through Apache Kafka, Apache Spark Structured Streaming for feature aggregation, ScyllaDB for ultra-low-latency feature lookups, a custom C++ inference engine with AVX-512 optimization, and C++ as the primary serving language. The paper documents the engineering decisions that enabled the SLA to be met under steady-state and peak load. A central decision was the migration of the scoring path from Python to C++ with AVX-512 SIMD intrinsics, which reduced p99 latency by approximately 4×. The paper also reports two production incidents observed over a two-year operational window and the post-incident changes that followed. Neither incident produced fraud leakage, illustrating that graceful degradation and observability are as important as raw performance to the operational viability of fraud detection systems.

Index Terms—fraud detection, real-time streaming, machine learning inference, low-latency processing, financial transactions, feature engineering

I. INTRODUCTION

Payment fraud is dynamic, adaptive, and economically consequential. Adversaries continuously probe for weaknesses in authorization systems, shifting their tactics as defenses harden, and the financial cost of undetected fraud is borne by the issuing bank, the merchant, and ultimately the customer in the form of higher fees and reduced trust. The systems that detect fraud in real time must therefore make accurate decisions on each individual transaction within an authorization window measured in tens of milliseconds, drawing on features that reflect the transaction's full historical context and applying models that capture the subtle patterns distinguishing legitimate from fraudulent activity.

This paper presents a case study of a production real-time fraud detection system operated at a major financial services company. The system processes between ten thousand and fifty thousand transactions per second, with peak rates reached during shopping seasons and concentrated geographic events. End-to-end latency from the arrival of a payment authorization request at the fraud service to the return of an accept-or-decline decision must fit within a two-hundred-millisecond service level agreement at the ninety-ninth percentile, a budget that must accommodate

feature retrieval, model inference, network round trips, and any downstream rule evaluation. The regulatory environment imposes additional constraints: the Payment Card Industry Data Security Standard (PCI-DSS) requires comprehensive audit trails, and anti-money-laundering and know-your-customer obligations require that decisions be explainable and that data access be logged.

The business stakes are substantial in both directions. Undetected fraud directly impacts customer confidence and the bank's revenue, since chargebacks and fraud the issuer absorbs fraud losses. False positives are equally consequential: declining a legitimate transaction frustrates the customer and may drive them to a competing payment method or competing card. The system must therefore optimize jointly for fraud detection rate and false positive rate, neither of which can be sacrificed for the other.

The paper centers around three research questions:

RQ1: What system architecture and component choices enable sub-200-millisecond fraud detection scoring at tens of thousands of transactions per second while maintaining feature freshness and model consistency?

RQ2: How do language and runtime choices (Java versus Python), connection pooling, and batching optimizations impact end-to-end latency and throughput in streaming fraud scoring pipelines?

RQ3: What operational patterns, incidents, and failure modes emerge in production fraud systems, and how should teams design for resilience and observability?

The remainder of the paper proceeds as follows. Section 2 reviews related work on fraud detection, streaming systems, and ML model serving. Section 3 describes the system architecture. Section 4 documents the feature engineering strategy and the rationale for the chosen feature windows. Section 5 explains the migration from Python to C++ for the scoring layer and reports the empirical performance comparison using processor-level optimization. Section 6 details the model inference pipeline built with custom C++ inference engine and AVX-512 vectorization. Section 7 describes the ScyllaDB caching and connection pooling optimizations that produced the largest single latency improvement in the system's history. Section 8 reports the two production incidents observed over a two-year window, including their root causes, resolutions, and the lessons drawn from them. Section 9 discusses monitoring and observability practices. Section 10 summarizes operational practices and SLA management. Section 11 compares the chosen design against alternative architectures. Section 12 presents lessons learned and best practices, Section 13 identifies future work, and Section 14 concludes. The principal contributions of the paper are an end-to-end documentation of a production fraud detection architecture with concrete latency budgets and component choices, an empirical comparison of Python and C++ for low-latency model serving, a detailed account of two production incidents with their root causes and resolutions, and a set of operational lessons applicable to other latency-critical payment processing ML systems.

II. RELATED WORK

Researchers have sustained fraud detection as an area of research for several decades, predating the modern machine learning era. Early statistical methods exploited regularities in legitimate transaction distributions, including the application of Benford's law to detect synthetic data fabrication. Rule-based fraud systems formalized

expert knowledge as configurable predicates and dominated production deployment through the 1990s and into the 2000s. The transition to machine-learning-based fraud detection accelerated in the 2010s as commodity computing made it economical to train large models on historical transaction data and as feature stores made it possible to serve those models with fresh inputs at request time [5], [6], [7].

The general problem of streaming anomaly detection researchers have studied in the data mining literature [9] and embodied in production systems including Yahoo S4 and Twitter Heron [14]. Apache Kafka [13] has become the dominant message broker for high-throughput event ingestion in financial systems, and Spark Structured Streaming [2] is widely used for windowed feature aggregation in micro-batch mode. Apache Flink [8] offers an alternative streaming model with lower latency but a different operational profile. The choice between Spark and Flink for fraud feature computation typically turns on the team's existing operational expertise and on the latency requirements of the specific feature category.

Real-time machine learning inference researchers have studied as a systems problem in its own right. Clipper [11] introduced a low-latency prediction serving system with adaptive batching and model selection. TensorFlow Serving [15] generalized the pattern for TensorFlow models. Nexus [16] addressed batching and scheduling for GPU-accelerated inference. The Open Neural Network Exchange (ONNX) format provides a language-agnostic representation for trained models, enabling models trained in Python to be served in C++, Rust, or other high-performance runtimes without re-implementation. A custom C++ inference engine with AVX-512 optimization provides vectorized computation for fast model evaluation on modern CPU architectures, achieving sub-millisecond latency for tree-based models through processor-level parallelism.

Feature engineering specific to fraud detection researchers have studied along several dimensions. Transaction graph approaches model the relationships among cards, merchants, and devices to detect ring activity and money laundering patterns [1]. Temporal features capture velocity the rate of transactions on a card or against a merchant within a short window which is among the strongest signals for credit card fraud. Merchant categorization features encode prior knowledge about the relative risk of different merchant types. The combination of velocity, behavioral, and contextual features dominates production fraud detection systems.

The specific question of language choice for low-latency systems has received less academic attention than its operational importance warrants. Python's Global Interpreter Lock (GIL) limits its ability to exploit multi-core hardware for CPU-bound workloads, which constrains the throughput achievable from a single Python process. The Java Virtual Machine (JVM) supports true preemptive multithreading and produces deterministic bytecode performance after warmup. For inference workloads where each request involves CPU-bound model evaluation, the difference between the two runtimes can be substantial. Connection pooling and request batching are well-established techniques in the database and distributed systems literature [12] but are sometimes underapplied in ML serving contexts.

Incident response and chaos engineering [4] have become standard practices in operating large-scale distributed systems, and the application of these practices to financial workloads must fit within the combination of regulatory scrutiny and customer impact. The case study reported in this paper

draws on these practices and adds documentation of two specific incidents that may be useful as test cases for future chaos-engineering exercises.

3. Fraud Detection System Architecture

The fraud detection system comprises five principal components arranged in a streaming pipeline. The transaction event source is Apache Kafka, which receives payment authorization requests as they arrive from the upstream payment processing layer. Each Kafka topic partitions to support parallel consumption by downstream components, with three-way replication providing durability against broker failures. Spark Structured Streaming consumes the transaction stream, computes windowed aggregations grouped by card and by merchant, and writes the results into ScyllaDB as cached features. The feature service component reads cached features from ScyllaDB at request time, combines them with features computed from the inbound transaction itself, and presents a complete feature vector to the model scorer. The model scorer is a high-performance C++ service that loads a gradient-boosted tree model and produces a risk score for each transaction using the custom inference engine with AVX-512 optimization. The risk score returns to the transaction engine, which combines it with rule-based decisions to produce the final accept-or-decline determination. Each component requires design to operate within a latency budget that, in aggregate, fits inside the two-hundred-millisecond end-to-end SLA.

The system handles a workload of approximately ten thousand to fifty thousand transactions per second under normal operation, with peaks during shopping seasons and concentrated geographic events. The team sizes Kafka topic partitions to accommodate the peak rate with headroom, and downstream consumers (Spark Structured Streaming applications, the feature service, and the scorer) scale horizontally on Kubernetes to absorb load spikes. ScyllaDB runs as a replicated cluster with automatic failover, sized to hold the working set of feature data without exhausting memory under normal cardinality assumptions. The team deploys the C++ scorer as a containerized application on Google Kubernetes Engine (GKE) with between ten and thirty replicas depending on observed load.

The end-to-end request flow proceeds as follows. A transaction event arrives in Kafka and routes to the feature service, which retrieves the relevant cached features from ScyllaDB using a batched lookup. The feature service combines the cached features with point-in-time features derived from the inbound transaction (geographic distance from the prior transaction, temporal deviation from the customer's normal pattern) and constructs a feature vector. The vector goes to the C++ scorer, which invokes the custom inference engine with AVX-512 optimization to produce a risk score. The risk score, together with metadata identifying the model version and the features used, returns to the transaction engine within the latency budget. Throughout the flow, structured logs record the inputs and outputs of each component for downstream investigation and audit.

IV. FEATURE ENGINEERING FOR FRAUD DETECTION

The feature set used by the fraud detection model organizes into four categories: card features, merchant features, temporal features, and geographic features. Card features capture the recent activity of the card itself, including the count of transactions in the last ten minutes, the total amount spent in the same window, and the average transaction amount. These features

computation derives from the streaming transaction log and reflect velocity-based fraud signals, since legitimate cardholders rarely produce sudden bursts of high-frequency transactions outside specific contexts. Merchant features capture the customer's history with the merchant in question, the merchant's overall transaction profile, and category-specific spending patterns over a forty-five-day window. Temporal features capture the time of day and day of week of the transaction, together with the deviation from the customer's average transaction timing. Geographic features capture the distance from the customer's home address, the distance from the most recent transaction, and the frequency of country changes over a forty-five-day window.

The choice of feature windows reflects a tradeoff between freshness and stability. Velocity features use ten-minute windows because fraud activity typically manifests as a burst of transactions within a short interval; longer windows would dilute the signal, while shorter windows would produce noisy estimates with low statistical confidence. Behavioral features use sixty-day windows because customer behavior evolves slowly and a longer window provides a more stable baseline against which to measure deviation. The combination of short-window velocity features and long-window behavioral features is standard in the fraud detection literature and was confirmed empirically against the model's accuracy on held-out data.

Feature computation implements in Spark Structured Streaming. The streaming application reads from the card transactions topic, computes tumbling windows on a sixty-second cadence, and writes the resulting aggregations to ScyllaDB with a fifteen-minute TTL. The Delta Lake ACID semantics on the underlying tables ensure that Spark recovery after failure produces consistent feature values rather than partial aggregations [3]. The fifteen-minute ScyllaDB TTL balances cache hit rate against the cost of recomputing missing features; it is longer than the ten-minute velocity window because Spark refreshes the values more frequently than the TTL expiration. If a feature is missing from the cache because of TTL expiry, a ScyllaDB incident, or a Spark job delay the feature service falls back to a snapshot held in cold storage on Confluent Cloud with Tiered Storage on Google Cloud Storage, accepting modestly stale features in exchange for continuity of service. The fallback path triggers an alert to the on-call team.

Feature freshness in steady-state operation is between zero and sixty seconds, corresponding to the gap between successive Spark micro-batches. This degree of freshness is acceptable for fraud detection because fraud patterns generally develop on a ten-minute or longer scale. If the Spark job delay occurs for any reason, feature ages increase, and the feature service reports degraded freshness through the monitoring dashboard. Graceful degradation using older feature values rather than failing the request is the dominant operational strategy, supported by alerts that bring the underlying delay to the operator's attention without forcing a failure of the inference pipeline.

V. LANGUAGE AND RUNTIME CHOICE: JAVA VERSUS PYTHON

The initial implementation of the fraud scorer was written in Python, using XGBoost as the inference library and a simple HTTP server to expose the scoring endpoint. Python was chosen first for the obvious reasons: the data science team was familiar with the language, the model training pipeline was already Python-based, and rapid prototyping was a priority during early

development. The first production deployment processed traffic at approximately one hundred transactions per second per process, with per-request latency between forty and sixty milliseconds. Both numbers were too far from the target tens of thousands of transactions per second across a horizontally scaled cluster, with sub-five-millisecond model inference latency to be addressed by simple horizontal scaling alone.

Profiling identified two principal causes of the latency and throughput shortfall. The first was the Global Interpreter Lock (GIL): the Python process could not effectively utilize the multi-core servers it deployed on, because the GIL serializes the execution of Python bytecode across threads. The thread pool for the scoring service saturated at approximately four to eight threads despite the underlying hardware offering many more cores. The second was garbage collection overhead, which produced unpredictable latency tail behavior even after the average case was acceptable. Together, these limitations made it difficult to achieve the deterministic, low-tail-latency behavior that a fraud SLA requires.

The team migrated the scoring layer to C++. The trained model was exported from Python to ONNX format, which provided a language-agnostic interchange representation that could be loaded directly by the custom C++ inference engine. The C++ scorer runs with a thread pool of fifty to one hundred threads, exploiting SIMD vectorization with AVX-512 instructions. Each thread maintains its own inference context optimized for tree traversal and leaf predictions. Compiler optimization using aggressive compiler flags for CPU-specific features takes approximately one minute during initialization and is handled by health-check gating: a freshly deployed instance is not added to the load balancer pool until its inference latency has stabilized.

Table 1 summarizes the empirical performance comparison between the Python and Java implementations on the same hardware and model.

Table 1: Python vs Java scorer performance.

Metric	Python (XGBoost)
Per-request inference latency	40–60 ms
Throughput per process	~100 TPS
Effective concurrency	4–8 threads (GIL)
Tail latency predictability	GC-driven jitter
Latency improvement	Baseline
Throughput improvement	Baseline

The numerical comparison should not be read as a general indictment of Python; the limitations identified are specific to a CPU-bound serving workload that requires SIMD vectorization for maximum throughput. For training, exploratory data analysis, and offline batch scoring, Python remains an excellent choice. The migration to C++ applied only to the request-time serving layer; model training, model evaluation, and the surrounding data science workflows remained in Python and continue to benefit from the rich Python ML ecosystem. ONNX is the bridge that makes this division of labor practical: data scientists train models in Python and export them to ONNX, and the C++ serving layer loads and executes them using processor-level vectorization without ever importing Python code.

VI. MODEL INFERENCE WITH CUSTOM C++ ENGINE AND AVX-512

The fraud detection model is a gradient-boosted tree ensemble trained with XGBoost [10]. Gradient-boosted trees are well suited to fraud detection for several reasons: they handle

heterogeneous feature types (continuous, categorical, count) natively; they produce feature importance scores that support post-hoc explainability; they are fast to evaluate at inference time because tree traversal is a series of simple comparisons; and they tend to be robust to feature scaling and to small amounts of missing data. The model used in production has between fifty and two hundred trees, with tree traversal accounting for approximately two to three milliseconds of inference time per transaction.

ONNX serves as the model interchange format. After training, the XGBoost model is exported to ONNX through the standard converter, producing a language-agnostic file that can be loaded by the custom C++ inference engine. The C++ serving layer loads the file and creates a per-thread inference context that leverages AVX-512 instructions for tree traversal and leaf prediction. Contexts are reusable across requests, eliminating the per-request setup cost that would otherwise dominate latency. The inference latency breakdown for a single transaction is approximately 0.3 to 0.7 milliseconds for feature vector preparation (data gathering and serialization), 1 to 2 milliseconds for the model tree traversal with SIMD operations, and 0.3 to 0.7 milliseconds for result preparation, for a total per-transaction inference latency of under 2 milliseconds.

Batching applies selectively. For pure throughput optimization, accumulating fifty to one hundred requests into a single batch and submitting them together amortizes per-call overhead and dramatically increases throughput at the cost of additional latency for individual requests held in the batch buffer. For fraud detection, where the latency budget is two hundred milliseconds and the per-request inference cost is already under five milliseconds, the marginal benefit of batching proves small relative to the latency cost it introduces, and the system uses batching primarily for the ScyllaDB lookup path described in Section 7 rather than for the model inference itself.

Model serving deploys as a containerized Java application based on Spring Boot with an embedded Tomcat server. The Kubernetes deployment runs ten to thirty replicas depending on observed traffic, with an HPA configured against custom metrics. Teams mount model files from shared storage, and a model update teams perform as a blue-green deployment: the new model is loaded into a fresh pod set, we shift traffic to the new set after health checks pass, and the old set is drained gracefully. Canary rollouts route five percent of traffic to a new model for an observation period before the remaining traffic is shifted, providing a window in which a regression can be detected and rolled back.

VII. SCYLLADB CACHING AND CONNECTION POOLING OPTIMIZATION

The largest single latency improvement in the system's history came not from model optimization or hardware upgrades but from a set of changes to how the C++ scorer interacted with ScyllaDB. The initial implementation issued a separate ScyllaDB client call for each feature it needed, with a fresh connection per request. Connection pooling maintaining a set of pre-established connections to ScyllaDB and borrowing one per request rather than opening a new one did not apply. The observed per-request feature retrieval latency was forty to sixty milliseconds, dominating the per-request budget and consuming most of the two-hundred-millisecond SLA for what should have been a simple cache lookup.

Profiling identified two distinct causes. The first was network round-trip overhead. Each feature retrieval required a TCP and TLS handshake to establish the ScyllaDB connection, with five to ten milliseconds of fixed cost per handshake even on a same-region network. The second was the multiplication of round trips: with ten to twenty features per request, each fetched in a separate ScyllaDB call, the per-request total network time was twenty to sixty milliseconds even before any feature processing. The model inference itself, once the features were in hand, was not the bottleneck.

Three optimizations were applied. The first was connection pooling: a pool of twenty to fifty persistent connections was established and maintained across requests, eliminating the per-request handshake cost. Pool sizing was tuned to match the concurrent request count of the scoring service, which has fifty to one hundred threads. Twenty to fifty connections proved sufficient to keep contention low without consuming excessive resources on either the C++ side or the ScyllaDB side. The second optimization was batched lookups: instead of issuing ten to twenty individual query operations, the scorer issues a single ScyllaDB batch request that retrieves all features for a given entity in one round trip. The number of round trips per request fell from ten to twenty down to one or two. The third was an in-application LRU cache holding approximately one hundred thousand recently accessed entities in memory, occupying approximately fifty megabytes of heap. For hot entities, the LRU cache absorbs eighty to ninety percent of the lookup load and produces sub-tenth-of-a-millisecond latency for in-memory hits.

The combined effect of the three optimizations reduced per-request feature retrieval latency from forty to sixty milliseconds down to under one millisecond. Listing 1 shows a sketch of the connection pool configuration.

Listing 1: ScyllaDB connection pool configuration sketch (C++ client-style).

```
JedisPoolConfig poolConfig = new
JedisPoolConfig();
poolConfig.setMaxTotal(50); // pool size
poolConfig.setMaxIdle(30); // idle
ceiling
poolConfig.setMinIdle(10); // pre-warmed
connections
poolConfig.setTestOnBorrow(false); //
skip per-borrow ping
poolConfig.setBlockWhenExhausted(true);
poolConfig.setMaxWaitMillis(50); // give
up fast

JedisPool pool = new JedisPool(poolConfig,
host, port, 200, password);

// Per-request usage
try (Jedis jedis = pool.getResource()) {
 Pipeline pipeline = jedis.pipelined();
 List<Response<String>> responses =
featureKeys.stream()
 .map(pipeline::get)
 .collect(Collectors.toList());
 pipeline.sync();
 return
responses.stream().map(Response::get).col
lect(...);
}
```

Table 2 summarizes the latency breakdown before and after optimization.

Table 2: Per-request latency breakdown before and after optimization.

Component	Before
Connection establishment	5–10 ms (handshake)
Feature retrieval round trips	10–20 (one per feature)
Total feature retrieval	40–60 ms
In-app LRU cache hit rate	
Model inference (ONNX)	2–3 ms
Per-request total (cache hit)	~60 ms

The lesson from the optimization sequence is that network I/O, not computation, was the dominant latency source in the original design. Profiling the actual production workload identified ScyllaDB as the bottleneck rather than the model inference, contradicting the initial assumption that model evaluation would dominate. Connection pooling and request batching are textbook techniques in distributed systems engineering, but they are sometimes underapplied in ML serving contexts where the team's attention naturally focuses on model performance. The forty- to sixty-fold latency reduction from these changes was larger than any improvement that could plausibly have come from algorithmic changes to the model itself.

VIII. PRODUCTION INCIDENTS AND OPERATIONAL RESILIENCE

Two production incidents were observed over a two-year operational window. Both were resolved without producing fraud leakage that is, neither resulted in fraudulent transactions being authorized when they should have been declined and both produced enduring improvements to the system's design and operational procedures. This section documents each incident in detail.

8.1 Incident 1: ScyllaDB Memory Exhaustion

The first incident occurred when ScyllaDB memory usage, which had been creeping upward over a period of weeks, crossed the critical threshold during a busy afternoon. By 3 PM, the memory utilization on the primary ScyllaDB instance had reached ninety percent. The LRU eviction policy began evicting feature keys at an accelerating rate. By 5 PM, memory was exhausted. ScyllaDB rejected writes from the Spark feature aggregation jobs. The fraud scorer continued to function but could no longer retrieve fresh features for many entities. The feature service fell back to its cold storage path and began serving features that were up to thirty-six hours old. End-to-end latency on the affected requests spiked from the normal five-to-ten-millisecond range to approximately six hundred milliseconds, breaching the SLA. The on-call team detected the spike via the latency dashboard and the cache miss rate alert, identified the root cause within fifteen minutes, and restarted ScyllaDB with an increased memory allocation. The total duration of the incident from initial breach to resolution was approximately eighteen minutes.

Root cause analysis identified two contributing factors. The first was a substantial underestimate of feature cardinality. The ScyllaDB instance the team sized for an estimated working set of approximately ten million unique entities, based on assumptions made during initial deployment about the number of active cards and merchants. The actual working set in production was approximately two hundred million entities, twenty times the estimate, driven by long-tail merchants and infrequently active cards that the original sizing analysis had not fully accounted for. The second contributing factor was that the default ScyllaDB maxmemory policy teams had not tuned for the fraud detection use case; the LRU eviction was triggering on the wrong key set under memory pressure.

Critically, despite the latency breach, no fraud did not occur during the incident. The fraud scorer continued to produce scoring decisions, drawing on stale features rather than failing the requests outright. The model's decision boundary remained reasonable on the older features, and the actual fraud detection rate during the eighteen-minute window was within normal bounds when measured ex post. Customer impact was limited to

slower-than-normal transaction processing. The graceful degradation strategy worked as designed.

The remediation increased the ScyllaDB memory allocation from thirty-two gigabytes to sixty-four gigabytes, configured an explicit allkeys-lru maxmemory-policy, and added monitoring with an alert at the eighty percent memory utilization threshold and an automatic graceful-degradation trigger at the ninety-five percent threshold. A runbook was added to the on-call documentation describing the symptoms, the diagnostic steps, and the standard remediation. The lesson is that capacity assumptions made during initial deployment must be revisited as actual production data becomes available, and that monitoring on cache memory utilization is at least as important as monitoring on cache hit rate.

8.2 Incident 2: Code Regression and False Positive Spike

The second incident occurred when a new model version deployed into production. The new model the training process completed on a larger dataset and had passed the offline evaluation suite with metrics comparable to the existing model. Within five minutes of deployment, however, the false positive rate began to climb sharply. Within thirty minutes, the false positive rate had reached approximately five times the baseline, with legitimate transactions being declined at unusual rates and customer support tickets accumulating quickly. The deployment was rolled back to the previous model version, and the false positive rate returned to normal under 45 seconds.

Root cause analysis identified a feature engineering bug in the training pipeline. One of the input features had been computed incorrectly during training, with a definition that subtly diverged from the production serving implementation. The bug was present in both the training set and the test set used for offline evaluation, so the offline metrics looked normal the model performed well on data that contained the same bug as the data it was trained on. In production, however, the serving pipeline computed the feature correctly, and the resulting input distribution differed from the distribution the model the training process completed on. The model's decision boundary, fit to the buggy training data, no longer aligned with the correct production inputs, and the result was a sharp shift in classification behavior that manifested as the false positive spike.

As with the first incident, no fraud did not occur during the incident window. The new model was over-aggressive rather than under-aggressive: it was declining too many legitimate transactions, but it was still declining the actual fraudulent ones. The customer impact was real and substantial, but the security posture of the system was preserved. The remediation introduced two new safeguards. The first was shadow deployment: new models are now run in parallel with the production model for forty-eight hours before any traffic routes to them, with both models scoring the same transactions and the resulting fraud detection rates compared. The second was canary deployment: after the shadow phase, the new model receives five percent of production traffic for an additional observation period of approximately four hours, during which the fraud detection rate, false positive rate, and latency are compared between the canary cohort and the control cohort. Automated alerts fire if the fraud detection rate deviates by more than ten percent from baseline. The lessons are that training-serving consistency must be verified end to end rather than within either environment in isolation, and that shadow and canary deployments are essential safeguards

for any model that affects customer-facing decisions.

8.3 Incident Summary

Table 3 summarizes the two incidents.

Table 3: Incident summary.

Incident	Duration	Symptom
ScyllaDB memory exhaustion	18 min	600 ms latency sp stale features
Model regression / false positives	30 min	False positive rate baseline

Two incidents in two years implies an average incident rate of approximately one per year, which is consistent with the operational maturity targets of comparable financial systems. Neither incident resulted in fraud leakage, which the authors attribute to the explicit graceful degradation paths in the design and to the monitoring practices that surfaced the incidents quickly enough for human intervention to be effective.

IX. MONITORING AND OBSERVABILITY

Observability is the foundation on which the fraud detection system's operational reliability rests. Several categories of metrics track continuously. End-to-end latency measures at the p50, p95, and p99 percentiles against the two-hundred-millisecond SLA, with breakdowns by card type, merchant category, transaction amount, and customer segment to support diagnosis of slow paths. Feature retrieval latency tracks separately, together with the ScyllaDB cache hit rate and the age of the cached features, with alerts firing when the hit rate drops below approximately ninety percent or when the feature age exceeds approximately thirty minutes. Model inference latency measures directly from the custom C++ inference engine with AVX-512 optimization instrumentation; values exceeding ten milliseconds are flagged as potentially indicating garbage collection pauses or thread contention.

Two categories of metrics deserve emphasis because they detect failures that latency monitoring alone would miss. The first is the fraud detection rate, estimated from the proportion of actual fraud caught by the model. The fraud detection rate is necessarily a delayed metric it depends on chargeback data, customer feedback, and manual review but a sustained drop in the rate is a strong signal that something has changed in the model or the upstream data. A target fraud detection rate of greater than ninety percent provides the operational floor. The second metric is the false positive rate, the proportion of legitimate transactions flagged as fraud. The target is below one percent, and an increase in the false positive rate is, as the second incident demonstrated, often the first observable symptom of a model regression.

Logging configures to record every transaction with its features, model score, and final decision. Log retention is ninety days, in line with regulatory requirements for fraud investigation. Logs index in Elasticsearch and can be queried by card identifier, merchant identifier, or time window, supporting both forensic investigation of suspected fraud and post-incident analysis of system behavior. Alerts fire on several conditions: a sudden increase in the fraud detection rate (greater than five times baseline, which often indicates a coordinated fraud attempt rather than a system problem); a latency spike (p95 above five hundred milliseconds); a cache miss rate above twenty percent; or a model score distribution shift in which the median score moves more than two standard deviations from the historical baseline.

A real-time dashboard displays the current latency percentiles, throughput, feature age, and fraud rate, together with historical trends over twelve hours and seven days. An incident view shows recent alerts and the current on-call rotation.

X. OPERATIONAL PRACTICES AND SLA MANAGEMENT

The system operates under a 99.9% availability SLA, which permits approximately 43.2 minutes of downtime or SLA breach per month. SLA breaches are defined as any outage or sustained latency spike affecting more than 0.1 percent of transactions. The on-call rotation provides primary coverage with a documented escalation path for incidents that cannot be resolved within the primary on-call's authority. Playbooks document the standard response for the most common incident classes, including ScyllaDB memory pressure, feature age violations, and model performance degradation. A post-incident review occurs within eighteen hours of any SLA breach, with the goal of identifying root causes and preventive measures rather than assigning blame.

Deployment practices design goals include minimize the risk of regression. Model deployments follow a shadow → canary → gradual rollout sequence. New models are deployed in shadow mode in the morning local time, where they receive copies of production traffic for thirty-six hours but their decisions are not used. Comparison of the shadow model's decisions against the production model's decisions detects most categories of regression before any customer-facing impact occurs. After shadow validation, the new model is promoted to canary mode, receiving five percent of production traffic for an additional four-hour window during which fraud detection rate, false positive rate, and latency are compared between the canary and control cohorts. If the fraud detection rate drops by more than ten percent, an automated alert fires; rollback is performed by the on-call engineer rather than automatically, to preserve human judgment over a critical security control.

Code deployments follow a feature-flag-based pattern. New optimizations or behavioral changes are deployed behind a feature flag, enabled on five percent of traffic, verified against monitoring metrics, and then enabled for the remaining traffic. Instant rollback through the feature flag mechanism is possible if any anomaly is detected. Disaster recovery supports by failover to a secondary region with a Recovery Time Objective of approximately five minutes (automated failover) and a Recovery Point Objective of approximately one minute (feature data replicated to the secondary region every sixty seconds).

XI. COMPARATIVE ANALYSIS AND DESIGN TRADEOFFS

Several alternative architectures were considered during the system's design. Each represents a defensible point in the design space, and the choice among them depends on the specific constraints of the deployment.

A synchronous request to a feature service for every scoring request would guarantee that features are always fresh, eliminating the staleness window introduced by the cache. The cost is an additional network round trip per request, adding five to ten milliseconds of latency on every transaction. For a sub-five-millisecond feature retrieval target, this cost is unacceptable. The chosen design accepts a small staleness window in exchange for the latency improvement that the cache provides.

A second alternative is to pre-compute features and embed them directly in the transaction event

payload as it flows through Kafka, eliminating the need for a separate feature service entirely. This pattern, sometimes called event enrichment, has the advantage that everything required for scoring is co-located with the transaction itself. Its disadvantages are that the feature payload grows substantially, schema evolution becomes difficult to manage, and the separation of concerns between feature engineering and model scoring is violated. The chosen design preserves the separation, allowing the feature team and the model team to iterate independently on their respective concerns.

A third alternative is to use approximate or sketch-based feature representations HyperLogLog for cardinality estimation, count-min sketches for frequency estimation which trade exact accuracy for substantial reductions in memory footprint and computation cost. The disadvantages are that approximation error reduces fraud detection accuracy by an unpredictable amount and that approximate features are difficult to explain to regulators who require deterministic decision criteria. For a deployment where regulatory audit is a hard requirement, exact feature computation is the safer choice. The chosen design uses exact features at the cost of higher memory and computation, accepting the operational complexity in exchange for explainability.

The chosen design therefore balances latency, accuracy, and maintainability without optimizing for any single dimension at the expense of the others. Its principal advantages are the clear separation of feature service and scoring service (which permits independent scaling and ownership), the use of standard open-source components (which reduces vendor lock-in and supports the broader ecosystem of operational tools), and the explicit graceful degradation strategy (which preserves fraud detection capability even when individual components fail).

XII. LESSONS LEARNED AND BEST PRACTICES

Several lessons emerge from the deployment that should be useful to other teams building latency-critical financial ML systems.

The first lesson is that latency is itself a feature, and it deserves the same engineering attention as accuracy. Every millisecond of feature retrieval or model inference latency consumes a portion of a finite SLA budget, and the team that profiles its actual production workload is better positioned to optimize the right thing than the team that optimizes from intuition. In the deployment described here, profiling revealed that ScyllaDB network I/O not model inference was the dominant latency source, and the largest single improvement came from connection pooling and request batching rather than from any algorithmic change.

The second lesson is that graceful degradation proves preferable to clean failure. The ScyllaDB memory exhaustion incident illustrated that a fraud detection system that continues to score requests with stale features is more useful than one that fails closed when the cache is unavailable. Designing for partial failure means defining behavior for each component's unavailability and accepting modest service degradation in exchange for continuity of the security function.

The third lesson is that training-serving consistency must be verified end to end. The model regression incident illustrated that a feature bug present in both the training and the test sets will produce a model that performs well on offline evaluation but fails in production. The mitigation is shadow deployment that compares the new model's behavior against the production model on actual production data, surfacing distribution shifts that offline evaluation cannot detect.

The fourth lesson is that observability enables fast incident response. Both incidents described in this paper were detected within minutes of onset because the relevant metrics latency, cache miss rate, false positive rate were instrumented and monitored. The fraud detection rate is the most important metric for measuring model quality and should be tracked alongside the more traditional latency and throughput metrics. Logs supporting per-transaction forensic investigation should be retained for at least the regulatory minimum and should be queryable along the dimensions card, merchant, time window that investigations actually use.

The fifth lesson is that incident response itself improves system resilience. The first incident motivated the addition of memory monitoring and the runbook that guides on-call engineers through the standard remediation. The second incident motivated the introduction of shadow deployment and canary rollouts, which eliminated an entire class of model regression risk. Each incident is an opportunity to improve the system's design, and post-incident reviews that focus on systemic improvements rather than individual blame produce better outcomes over time.

The sixth lesson concerns team structure. Separating the feature platform team from the model development team allows each to optimize its own concerns cache sizing, batch policies, TTL values for the platform team; feature definitions, model architecture, training data for the model team without coordination overhead on every change. Clear SLAs between the teams (features fresh within five minutes, served within one millisecond) define the interface and prevent boundary disputes.

The seventh lesson is that regulatory and business context drives architecture. Financial fraud prevention is an existential business concern, PCI-DSS compliance requires comprehensive audit trails, and false positives directly damage customer trust. The architecture must optimize jointly for fraud detection rate and false positive rate, and it must produce decisions that can be explained to regulators and customers alike. The combination of these constraints rules out some otherwise attractive design choices, including approximate feature computation and unexplainable model architectures.

XIII. FUTURE WORK

Several directions warrant further attention as the system matures. Real-time model retraining updating the model on a daily or near-daily cadence as new fraud patterns emerge would reduce the lag between novel adversarial behavior and the model's response to it. The current cadence is weekly to monthly batch retraining, which is sufficient for slow-evolving fraud patterns but may underperform against fast-adapting adversaries. Distributed feature computation pushed closer to the point of decision (edge inference at regional payment processors) could reduce network latency further, although it complicates the consistency guarantees that the central feature store currently provides. Graph-based anomaly detection that incorporates the structure of merchant networks and customer networks would capture relational fraud signals that the current per-transaction features miss [1]. Automatic feature engineering through deep learning or program synthesis would reduce the manual effort currently required for feature definition, although the explainability requirements of regulated environments place limits on how far this automation can be trusted. SHAP and similar post-hoc explainability techniques would support customer-facing explanations of declined transactions, addressing a frequent customer complaint. Finally, cross-

institution collaboration to share anonymized fraud signals across partner banks could improve detection accuracy for distributed fraud patterns that no single institution sees in isolation.

Feature Store Architecture Patterns

3.1 Online and Offline Feature Stores

The dominant architectural distinction in feature store design is between offline and online stores. The offline feature store holds the historical, batch-computed feature data used for model training and large-scale batch scoring. It is typically implemented over inexpensive object storage Amazon S3, Confluent Cloud with Tiered Storage on GCS, or Google Cloud Storage and the the system organizes data for efficient analytical scans rather than for low-latency point lookup. Latency for offline retrieval measures in seconds to minutes, and cost per byte remains low. The online feature store holds the same logical features in a representation optimized for low-latency point lookup at request time. It is typically implemented over an in-memory or memory-resident store such as ScyllaDB or DynamoDB, latency measures in single-digit milliseconds, and cost per byte is substantially higher. Maintaining consistency between the two stores so that a feature retrieved online matches the corresponding feature recorded in the offline history is a defining design challenge.

The two stores serve different consumers. The offline store is consumed primarily by training pipelines that produce model artifacts, and by batch scoring jobs that compute predictions on large sets of records. The online store is consumed by inference services that handle individual scoring requests with strict latency budgets. A feature store implementation must support both consumption modes with consistent semantics.

3.2 Push and Pull Patterns

Two patterns dominate the population of the online store. In the push pattern, a background computation job typically a Spark or Flink streaming or micro-batch job computes feature values from source data and writes them into the online store. The inference service then reads pre-computed values from the cache. In the pull pattern, the inference service computes (or fetches and caches) feature values at request time, either directly from source data or

by invoking a feature computation service. Push patterns minimize request-time latency at the cost of staleness equal to the push interval; pull patterns minimize staleness at the cost of higher request-time latency. Hybrid patterns combine the two, pushing the bulk of features on a regular cadence and computing a smaller delta at request time.

3.3 Hybrid Baseline-Plus-Delta Architecture

The case study presented in Section 4 uses a hybrid pattern that the authors describe as a baseline-plus-delta architecture. A Spark job running on a five-minute cycle computes baseline aggregations card spending totals over rolling windows, merchant transaction counts, customer status indicators and writes the results into ScyllaDB with a configured TTL. A consumer process running inside the inference service holds an in-memory delta map for each entity, populated by streaming updates and by Type 2 SCD (Slowly Changing Dimension) joins against historical entity state. At request time, the inference service issues a single ScyllaDB lookup for the baseline feature vector and aggregates it with the in-memory delta to produce the final feature vector. The architecture trades complexity for the ability to combine the freshness of stream-driven updates with the throughput and predictability of batched baseline computation.

3.4 The Multi-Layer Feature Pipeline

A useful conceptual model for the feature pipeline distinguishes four layers. The raw data layer holds the source records as they arrive from operational systems Kafka topics fed by change data capture, batch files, or direct API ingestion. The base feature layer holds aggregations and derivations computed from raw data on a regular cadence, persisted in both the offline and online stores. The derived feature layer holds compositions of base features, often produced at request time or on shorter cadences. The cached online feature layer holds the request-ready feature vectors served to inference clients. Each layer has its own consistency, latency, and cost properties, and the boundary at which a particular feature transitions from one layer to the next is a design decision driven by the model's freshness requirements and the operational cost of recomputation.

Implementation Case Study

4.1 Context and Stack

The deployment that grounds this paper is operated at a large payments processor. The platform team responsible for the feature store grew from seven to nineteen engineers over the development period and is organized into domain-aligned squads supporting fraud detection, payment risk assessment, and chargeback prediction, plus a platform squad that maintains the shared infrastructure. Feature consumers include more than thirty data science and ML engineering teams across the organization, each operating between five and fifty production models. The infrastructure stack uses Apache Kafka for event streaming, Apache Spark (both batch and Structured Streaming) for feature computation, Confluent Cloud with Tiered Storage on Google Cloud Storage for the offline data lake and historical feature storage, ScyllaDB for ultra-low-latency feature lookups, and Python and Scala for ETL and ML workloads.

4.2 Base Feature Refresh Job

The base feature refresh job is a Spark application orchestrated by Airflow on a five-minute cycle. The job reads recent partitions from the Delta Lake source tables, computes a configured set of aggregations and derivations card spending totals over rolling windows of one, seven, and thirty days; merchant transaction counts and average ticket sizes; customer activity indicators and writes the results into ScyllaDB through a pipeline of batched writes. Typical execution time for the job is two to three minutes, and the job's ScyllaDB writes are configured to use a TTL that, after the correction described in Section 5.2, is set to five minutes. The job produces approximately 50 distinct features across the four principal entity types (customer, card, merchant, transaction).

Listing 1 shows a sketch of the ScyllaDB key schema used by the job. The flat-key structure entity_type:entity_id:feature_name supports both single-key lookups and ScyllaDB pipeline-based batch retrievals.

Listing 1: ScyllaDB key schema and example values.

```
# entity_type : entity_id : feature_name
card:12345:spend_1d -> 412.55
card:12345:spend_7d -> 2891.10
card:12345:spend_30d -> 11240.75
card:12345:txn_count_1d -> 7
```

merchant:9876:avg_ticket -> 64.20

merchant:9876:risk_score -> 0.12

customer:55555:status -> ACTIVE

customer:55555:tenure_days -> 1287

4.3 In-Consumer Delta Computation

The consumer process that serves real-time inference requests does not rely on ScyllaDB alone. At process startup, the consumer subscribes to a small set of Kafka topics carrying transaction events and entity state changes. As events arrive, the consumer maintains an in-memory delta map that records the changes to each entity since the most recent baseline refresh. At request time, the consumer issues a single ScyllaDB lookup to retrieve the baseline feature vector, applies a Type 2 SCD join against the historical entity state to reflect the entity's status at the time of the request, and aggregates the baseline with any in-memory delta to produce the final feature vector returned to the model. The Type 2 SCD join must occur because financial entities have status changes a card may be reported lost, a customer may move address, a merchant may change risk classification and the model must score against the entity state that was current at the time of the underlying transaction, not at the time of inference. The aggregation step typically completes in under five milliseconds per lookup, with the ScyllaDB call accounting for approximately one millisecond and the SCD join and delta aggregation accounting for the balance.

4.4 Shared Python Feature Module

Training-serving consistency in the deployment is enforced through a shared Python module that defines feature computation functions used by both the offline training pipelines and the online serving consumers. The module is version-controlled in the platform's repository, and any change to a feature definition produces a versioned function name that downstream consumers must opt into. When a data scientist defines a new feature, the function is added to the shared module, and both the offline batch job that populates training datasets and the online consumer that serves inference requests import the same function. This shared-code approach addresses the training-serving skew problem identified by Sculley et al. [2015] by ensuring that the offline and online code paths are not merely conceptually

equivalent but literally the same code.

The shared module pattern has limits. It assumes that all consumers can share a Python runtime, which fails for inference services written in other languages. It also assumes that feature computation is expressible as pure functions over input data, which fails when computation depends on external state or on joins that the module cannot reproduce. For the deployment described here, both assumptions hold, and the shared module has prevented the bulk of training-serving consistency incidents that would otherwise be expected.

4.5 Operational Metrics

Table 1 summarizes the principal operational metrics observed during steady-state operation.

Table 1: Production operational metrics for the feature store.

Metric	Observed value	Notes
Per-key ScyllaDB lookup latency	0.5–1 ms	Pipeline batches achieve <2 ms for 40–60 keys
End-to-end feature retrieval (consumer)	<5 ms	Includes SCD join + delta aggregation
Cache hit rate	≥95%	Misses driven by cold start and TTL boundary
Base refresh job duration	2–3 minutes	5-minute Airflow cycle
Restart staleness window	5 minutes	Until next baseline refresh
ScyllaDB memory footprint	5–10 GB	Approximately 10M entities
Availability	99.9%	Replicated ScyllaDB, failover <30 s
Feature coverage	~50 features	Across customer/card/merchant/transaction

The cache hit rate of approximately ninety-five percent reflects steady-state operation; the misses are dominated by cold-start effects when consumers restart and by TTL boundary conditions discussed in Section 5.2. The ScyllaDB memory footprint of five to ten gigabytes corresponds to a working set of approximately ten million entities across the four entity types and is well within the memory budget of a single replicated ScyllaDB cluster.

Feature Freshness and Consistency Challenges

5.1 Freshness on Restart

The first failure mode encountered in production was a freshness regression that occurred whenever a consumer process restarted.

Restarts are routine: deployments, version upgrades, autoscaling events, and recovery from crashes all trigger a restart of the consumer process. When a consumer restarts, the in-memory delta map is lost, and the first inference requests after restart are served with feature values that reflect only the most recent baseline refresh, missing any updates that arrived between the last baseline and the restart. The observed staleness window is bounded by the baseline refresh interval five minutes in the deployment described here but during that window the model produces feature vectors that differ from those it would have produced under steady-state operation.

The team considered three possible mitigations. The first was to checkpoint the delta map to durable storage on a frequent cadence and reload it on restart. This option was rejected because it added complexity and a new failure surface to a hot path that the team wanted to keep simple. The second was to increase the baseline refresh frequency, reducing the staleness window at the cost of higher Spark and ScyllaDB load. This option was rejected because the cost increase did not justify a marginal staleness reduction for a class of events (restarts) that occurs only intermittently. The third option, which was adopted, was to accept the staleness as an operational reality, document it in the ML contracts published to consuming teams, and add monitoring to detect anomalous behavior immediately after restart. The lesson is that not all forms of feature staleness require engineering remediation; some can be absorbed into the SLA and managed through transparency and monitoring.

5.2 The One-Hour TTL Mistake

The second failure mode was a configuration error in the initial ScyllaDB TTL setting. The first version of the deployment configured a one-hour TTL on cached features, on the assumption that hourly cache invalidation would balance freshness against cache miss cost. In production, a subtle inconsistency emerged: features cached near an hour boundary expired during the subsequent hour, and models scoring at different points within the hour saw inconsistent feature values for the same entity. Two models scoring the same transaction within minutes of each

other could produce different predictions because one model retrieved a feature value just before TTL expiry and the other retrieved a freshly recomputed value just after.

Diagnosis took two to three days of root cause analysis because the symptom inconsistent scoring across nearby requests did not immediately implicate the TTL configuration. The resolution was to reduce the ScyllaDB TTL to five minutes, matching the baseline refresh cycle so that cached values are refreshed before they have a chance to expire under load. The cache miss rate increased modestly under the new configuration, but the increase was negligible for end-to-end latency because the Spark refresh job is fast enough to repopulate missing keys quickly. The lesson is that TTL configuration should be tied to the refresh interval and to the model's tolerance for staleness, not chosen as an arbitrary engineering default.

5.3 Training-Serving Consistency at Scale

The third consistency challenge is organizational rather than technical. As the number of consuming teams grew from approximately five to more than twenty, feature definitions began to diverge across teams. One team's fraud model required a ten-day card history window, while another team's risk model required a fourteen-day merchant average over a different aggregation. The shared Python module approach described in Section 4.4 prevented training-serving skew within a single feature definition, but it did not prevent the proliferation of subtly different feature definitions across teams. The mitigation was to introduce explicit feature versioning base_features_v1, base_features_v2, and so on and a metadata registry that documented each version's definition, owner, and consumers. The cost of this discipline is reduced flexibility for individual teams; the benefit is the ability to reason about which version of a feature any given model is consuming and to enforce backward compatibility when versions are deprecated.

Table 2 summarizes the failure modes and their mitigations.

Table 2: Failure modes and mitigations.

---------- ---------- ---------- ---------- ---------- Failure mode Trigger

Impact	Mitigation
Restart staleness	Consumer restart loses in-memory deltas 5-min staleness in features Accept; document in SLA; monitor TTL boundary inconsistency 1-hour TTL expiry under load Inconsistent scoring across requests Reduce TTL to 5 min (refresh-aligned) Feature definition drift Multiple teams diverge on definitions Training-serving skew across models Versioned feature module + registry ScyllaDB memory exhaustion Insufficient eviction tuning Cache grew to 100+ GB Size-based eviction + heap monitoring Connection exhaustion Naive ScyllaDB client under load Latency spikes, request blocking Connection pool sizing + idle timeout ScyllaDB unavailability Failover or network partition Loss of online serving Stale-flag policy + alert; degraded SLA

ScyllaDB Caching Patterns and Sub-Millisecond Latency

ScyllaDB teams selected as the online feature store for several reasons. The first is performance: per-key lookup latency in ScyllaDB is typically 0.5 to 1 millisecond on modern hardware in a same-region deployment, well within the budget for sub-five-millisecond feature retrieval. The second is operational familiarity: financial services engineering teams are generally comfortable operating ScyllaDB, which has been a standard caching technology in the industry for over a decade. The third is built-in TTL and eviction support, which simplify the management of feature freshness compared to a general-purpose key-value store that requires application-layer expiration logic. The fourth is cost: at scale, ScyllaDB memory is substantially cheaper than the equivalent capacity in a transactional database, and the access pattern of feature retrieval (point lookups, no transactions, no complex queries) maps well onto ScyllaDB's strengths.

Several design patterns emerged from the deployment. The flat key structure entity_type:entity_id:feature_name described in Section 4.2 supports both individual lookups and ScyllaDB pipeline-based batch retrievals. Aggregated feature vectors storing all features for a given entity and model under a single key reduce the number of round trips required for a complete feature retrieval. Batch

lookups via ScyllaDB pipeline can retrieve forty to sixty keys in a single network round trip with end-to-end latency under two milliseconds, which proves critical for models that consume many features per scoring request. An in-application LRU cache holding ten thousand to one hundred thousand recently accessed entities further reduces ScyllaDB traffic for hot entities and is particularly effective when the access pattern follows a power-law distribution, as is typical for fraud detection workloads in which a small number of high-velocity merchants account for a large fraction of transactions.

Three operational failures involving ScyllaDB were documented during the deployment. The first was a memory exhaustion incident in which the cache grew to over one hundred gigabytes because the eviction policy teams had not tuned. The remediation introduced a size-based eviction policy and monitoring on heap pressure. The second was a connection exhaustion incident caused by a naive ScyllaDB client design that opened a new connection per request. Under load, the connection pool was overwhelmed and request latency spiked. The remediation tuned the connection pool size to the concurrent request count and configured an idle connection reclamation policy. The third class of incidents involved ScyllaDB unavailability failover, network partitions, or operator error and required a policy decision about how the inference service should behave when the cache is unavailable. The chosen policy serves predictions with a stale-flag header and triggers an operational alert, accepting an SLA degradation during the incident rather than blocking inference entirely.

Organizational Scaling: From Simple to Complex

The deployment evolved through three distinct organizational phases, each with characteristic patterns and pain points.

In the early phase, the platform served approximately five models for a single team. Feature curation was performed manually in Python notebooks, all features lived in a single ScyllaDB namespace, the TTL was uniform across features, and the feature definitions were tightly coupled to the model code that consumed them. There was no metadata registry and no formal feature versioning. The architecture was

simple, the operational burden was low, and the coordination cost was negligible because all consumers were members of the same team.

In the growth phase, the platform served approximately twenty models across five to eight teams. Feature requests began to multiply, with teams demanding custom windows, custom aggregation frequencies, and team-specific transformations of shared base features. A platform squad was formed to centralize the feature store, separating responsibility for the underlying infrastructure from responsibility for the feature definitions themselves. Feature versioning and a metadata registry were introduced. The Spark ETL was refactored into a configuration-driven framework using Scala traits to define feature classes. Service-level agreements on feature freshness and availability were formalized.

In the scale phase, the platform served more than fifty models across more than twenty teams in multiple business domains. Several pressures combined to break the simpler approaches that had worked at smaller scale. Feature definitions could no longer be unified across all teams because domain-specific needs a fraud team's view of card velocity differs materially from a credit team's view diverged in ways that could not be papered over. Cache eviction became non-trivial because the working set exceeded the convenient memory budget of a single ScyllaDB cluster, and choices had to be made about which features to retain in cache. Coordination overhead between the platform team and the consuming teams began to consume a measurable share of platform engineering time, often exceeding the marginal benefit that any individual team derived from the shared infrastructure. The decision point that the team faced was whether to invest in a more sophisticated feature marketplace (with explicit registration, discovery, and lineage tracking) or to silo features by domain and accept some duplication of effort. The empirical observation from the deployment is that the simple shared approach works well up to approximately five models per consuming team and approximately ten consuming teams; beyond those thresholds, the marginal cost of additional standardization exceeds its benefit, and a more deliberate

organizational structure must occur.

Comparison with Alternatives

Three alternative architectures for online feature serving merit comparison with the ScyllaDB-based hybrid approach described above.

In-database online serving uses a transactional database either a row-oriented relational system or a key-value store with strong consistency guarantees as the primary online feature store. The principal advantage is strong consistency: a feature written by one transaction is visible to all subsequent reads, and ACID guarantees eliminate a class of race conditions that can occur with eventually consistent caches. The principal disadvantages are higher latency (typically ten to fifty milliseconds per lookup, depending on the database and the query pattern), higher cost per feature served, and greater operational complexity than a memory-resident cache. In-database serving is appropriate for use cases where strict consistency takes precedence over latency, such as regulatory reporting workflows where the audit trail must reflect a single canonical view of feature state.

OLAP databases including Apache Druid and Apache Pinot have been used as feature serving infrastructure in some organizations. Their principal advantage is flexible aggregation: arbitrary windowed aggregations can be computed at query time, eliminating the need to pre-compute and cache every feature. Their principal disadvantages are eventual consistency, higher operational overhead than a key-value cache, and a query model that is more general than required for point lookups. OLAP-based feature serving is appropriate for exploratory feature engineering and for analytical workloads, but it is typically not the right choice for deterministic real-time scoring.

Stateless microservices that compute features on demand at request time avoid the cache entirely. Each scoring request triggers a feature computation service that fetches the necessary source data, applies the feature logic, and returns the result. The principal advantage is freshness: the features always reflect the latest source data. The principal disadvantages are cache warmup on boot, higher per-request cost, and the need to serialize feature computation behind an additional network hop. Stateless feature

computation is appropriate when source data proves small enough to fetch quickly and when staleness tolerance is very low.

The ScyllaDB-backed hybrid approach occupies a middle position in this design space. It accepts a small amount of staleness (five minutes for baseline features, near-real-time for in-memory deltas) in exchange for sub-millisecond serving latency, and it accepts the operational overhead of two coordinated systems (Spark for refresh, ScyllaDB for serving) in exchange for the predictability that comes from pre-computing the bulk of feature work outside the request path.

Lessons Learned and Operational Guidance

Several lessons from the deployment are worth highlighting for practitioners building similar systems.

The first lesson is to design for graceful degradation. The feature store will be unavailable at some point during its operational life, and the inference service must have a defined behavior for that case. Serving with a stale-flag header and triggering an alert proves preferable to blocking the inference request entirely; the downstream consumers of inference can decide for themselves whether to honor stale predictions. Similarly, when a feature is missing from the most recent batch refresh, falling back to the previous day's value with a flag for downstream analytics proves preferable to returning a null value that may produce a model crash.

The second lesson is to invest in observability early. The metrics that matter for a feature store are the cache hit rate, the feature retrieval latency at p50, p95, and p99, and the feature age (the time elapsed since the underlying data was last refreshed). Alerts should fire when the cache miss rate exceeds approximately ten percent, when p95 latency exceeds approximately two milliseconds, and when feature age exceeds approximately twice the configured TTL. A team-facing dashboard exposing these metrics gives consuming teams visibility into the freshness and availability of the features they depend on.

The third lesson is to version everything. Feature definition versions, schema versions, and cache key format versions should all be explicit and managed. Blue-green deployment with a dual-

write period accommodates cache key format changes without downtime, and backward compatibility for at least two versions accommodates the lag between feature platform updates and consuming team updates.

The fourth lesson is to separate concerns between the platform team and the domain teams. The platform team owns the infrastructure, the caching layer, and the consistency guarantees. The domain teams own the feature definitions, the SLAs, and the model development lifecycle. The interface between the two is the feature registry API, the versioned Python module, and the compliance hooks for audit and access control. Clear ownership boundaries reduce coordination overhead and prevent both teams from making decisions that should belong to the other.

The fifth lesson is that TTL and eviction policy tuning is domain-specific and must not rely on engineering defaults. Shorter TTLs produce fresher features at the cost of higher cache miss rates and higher compute load on the refresh job. Longer TTLs produce higher hit rates at the cost of staler scoring. For financial services workloads where stale features can produce regulatory exposure, the bias should be toward freshness, with TTLs in the single-digit-minutes range. The cost of a slightly higher cache miss rate is almost always negligible relative to the cost of a stale-feature compliance incident.

The sixth lesson is to plan for deployment and restart chaos. Consumer restarts are routine, and the staleness window they introduce is a normal operating condition rather than a fault. For most use cases, the staleness is acceptable and can be absorbed into the SLA. For use cases where it is not acceptable, delta checkpointing or a persistent feature index can be implemented at additional cost. The decision should be driven by an explicit cost-benefit analysis rather than by reflexive engineering conservatism.

XIV. CONCLUSION

Real-time fraud detection in payment authorization is a demanding combination of latency, accuracy, and operational reliability requirements that few other production ML applications match. The system described in this paper event-driven ingestion through Apache Kafka, Apache Spark Structured Streaming for feature aggregation, ScyllaDB for online feature caching, custom C++ inference engine with AVX-512 optimization for model inference, and Java as

the primary serving language satisfies a two-hundred-millisecond end-to-end SLA at tens of thousands of transactions per second, and has done so reliably across two years of production operation interrupted by only two minor incidents. Neither incident resulted in fraud leakage, demonstrating that the explicit graceful degradation paths and the monitoring instrumentation described in Sections 8 and 9 do their intended work.

The principal lessons are that latency is a feature deserving the same engineering attention as accuracy, that the bottleneck in a real production workload is rarely where the team's intuition expects it (in this case ScyllaDB network I/O rather than model inference), that connection pooling and request batching often produce larger improvements than algorithmic optimization, that graceful degradation under partial failure proves preferable to clean failure, and that shadow deployment combined with canary rollouts is the only reliable way to catch model regressions before they affect customers. Future work in real-time retraining, graph-based fraud signals, and explainability will extend the system in directions that the current architecture supports but does not yet exploit. As payment fraud continues to evolve and as the models defending against it become more sophisticated, the operational discipline required to run these systems well will remain at least as important as the modeling techniques themselves.

REFERENCES

[1] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: A new model and architecture for data stream management. The VLDB Journal, 12(2), 120–139.

[2] Akoglu, L., Tong, H., and Koutra, D. (2015). Graph based anomaly detection and description: A survey. Data Mining and Knowledge Discovery, 29(3), 626–688.

[3] Armbrust, M., Das, T., Davidson, A., Ghodsi, A., Or, A., Rosen, J., Stoica, I., Wendell, P., Xin, R., and Zaharia, M. (2018). Structured Streaming: A declarative API for real-time applications in Apache Spark. Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, 601–613.

[4] Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., et al. (2020). Delta Lake: cloud object store table format with ACID guarantees. Proceedings of the VLDB Endowment, 13(12), 3411–3424.

[5] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. (2016). Chaos engineering. IEEE Software, 33(3), 35–41.

[6] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency Control and Recovery in Database Systems. Addison-Wesley.

[7] Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. (2016). Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media.

[8] Bolton, R. J., and Hand, D. J. (2002). Statistical fraud detection: A review. Statistical Science, 17(3), 235–255.

[9] Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5–32.

[10] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4).

[11] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. (2017). State management in Apache Flink: Consistent

stateful distributed stream processing. Proceedings of the VLDB Endowment, 10(12), 1718–1729.

[12] Carcillo, F., Le Borgne, Y. A., Caelen, O., and Bontempi, G. (2018). Streaming active learning strategies for real-life credit card fraud detection: Assessment and visualization. International Journal of Data Science and Analytics, 5(4), 285–300.

[13] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. ACM Computing Surveys, 41(3), 1–58.

[14] Chandy, K. M., and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, 3(1), 63–75.

[15] Chen, T., and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 785–794.

[16] Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., and Stoica, I. (2017). Clipper: A low-latency online prediction serving system. NSDI, 613–627.

[17] Dal Pozzolo, A., Boracchi, G., Caelen, O., Alippi, C., and Bontempi, G. (2018). Credit card fraud detection: A realistic modeling and a novel learning strategy. IEEE Transactions on Neural Networks and Learning Systems, 29(8), 3784–3797.

[18] Dean, J., and Barroso, L. A. (2013). The tail at scale. Communications of the ACM, 56(2), 74–80.

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 41(6), 205–220.

[20] Fawcett, T., and Provost, F. (1997). Adaptive fraud detection. Data Mining and Knowledge Discovery, 1(3), 291–316.

[21] Fitzpatrick, B. (2004). Distributed caching with Memcached. Linux Journal, 2004(124), 5.

[22] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. Annals of Statistics, 29(5), 1189–1232.

[23] Gilbert, S., and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 33(2), 51–59.

[24] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. (2014). The Java Language Specification: Java SE 8 Edition. Addison-Wesley.

[25] Helland, P. (2009). Life beyond distributed transactions: An apostate's opinion. CIDR, 132–141.

[26] Hwang, J. H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., and Zdonik, S. B. (2005). High-availability algorithms for distributed stream processing. Proceedings of the 21st International Conference on Data Engineering (ICDE), 779–790.

[27] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. Advances in Neural Information Processing Systems, 30, 3146–3154.

[28] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[29] Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. Proceedings of the NetDB Workshop.

[30] Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J. M., Ramasamy, K., and Taneja, S. (2015). Twitter

Heron: Stream processing at scale. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 239–250.

[31] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7), 558–565.

[32] Lundberg, S. M., and Lee, S. I. (2017). A unified approach to interpreting model predictions. Advances in Neural Information Processing Systems, 30, 4765–4774.

[33] Marz, N., and Warren, J. (2015). Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning.

[34] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. (2013). Scaling Memcache at Facebook. NSDI, 13, 385–398.

[35] Olston, C., Fiedel, N., Gorovoy, K., Harmsen, J., Lao, L., Li, F., Rajashekhar, V., Ramesh, S., and Soyke, J. (2017). TensorFlow-Serving: Flexible, high-performance ML serving. arXiv preprint arXiv:1712.06139.

[36] Phua, C., Lee, V., Smith, K., and Gayler, R. (2010). A comprehensive survey of data mining-based fraud detection research. arXiv preprint arXiv:1009.6119.

[37] Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. (2018). Data lifecycle challenges in production machine learning: A survey. ACM SIGMOD Record, 47(2), 17–28.

[38] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., and Dennison, D. (2015). Hidden technical debt in machine learning systems. Advances in Neural Information Processing Systems, 28, 2503–2511.

[39] Shen, H., Chen, L., Jin, Y., Zhao, L., Kong, B., Philipose, M., Krishnamurthy, A., and Sundaram, R. (2019). Nexus: A GPU cluster engine for accelerating DNN-based video analysis. Proceedings of the 27th ACM SOSP, 322–337.

[40] Sridharan, C. (2018). Distributed Systems Observability. O'Reilly Media.

[41] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. ACM SIGMOD Record, 34(4), 42–47.

[42] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm @Twitter. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 147–156.

[43] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th Annual Symposium on Cloud Computing, 1–16.

[44] Wang, G., Chen, L., Dikshit, A., Gustafson, J., Chen, B., Sax, M. J., Roesler, J., Blee-Goldman, S., Cadonna, B., Mehta, A., Madan, V., and Rao, J. (2019). Consistency and completeness: Rethinking distributed stream processing in Apache Kafka. Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data.

[45] West, J., and Bhattacharya, M. (2016). Intelligent financial fraud detection: A comprehensive review. Computers & Security, 57, 47–66.

[46] Whitrow, C., Hand, D. J., Juszczak, P., Weston, D., and Adams, N. M. (2009). Transaction aggregation as a strategy for credit card fraud detection. Data Mining and Knowledge Discovery, 18(1), 30–55.

[47] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. Proceedings of the 24th ACM Symposium on Operating Systems Principles, 423–438.

[48] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache Spark: A unified engine for big data processing. Communications of the ACM, 59(11), 56–65.

[49] Armbrust, M., et al. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Proceedings of CIDR 2021.