International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 6 Issue 1 PP 158-167, 2025
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V6I1P119

Original Article

Streaming Feature Stores and Real-Time ML Inference on Cloud-Native Infrastructure

Sivadeep Katangoori
Solutions Architect at Metanoia Solutions Inc., USA.

Received On: 09/03/2025 ***Revised On: 20/03/2025*** ***Accepted On: 24/03/2025*** ***Published On: 30/03/2025***

Abstract - *In the present day's fast-changing digital world, actual time machine learning (ML) is essential for making smart, responsive apps like fraud detection, recommendation systems, dynamic pricing & personalized user experiences. The feature store is the most important part of this change. It makes sure that ML models get the information that is timely, consistent & of high quality. This study investigates the evolution of feature stores from traditional batch pipelines to modern streaming architectures that provide real-time inference. As businesses increasingly embrace low-latency decision-making, the need for streaming feature storage has surged, facilitating the rapid production, modification, and transmission of features within milliseconds. We look at how cloud-native technology, such as Kubernetes, serverless platforms, and event-driven architectures, makes it possible to build real-time machine learning systems that can handle a lot of data, are reliable, and don't cost a lot of money. We want to provide a complete picture of how streaming feature stores function in practice, the architectural frameworks that support them, and the trade-offs that come with them. We demonstrate, via a practical case study, the use of a cloud-native stack to develop a real-time machine learning pipeline capable of ingesting and transforming substantial streaming data for low-latency inference. The case study clarifies the challenges faced, the implemented solutions, and the key performance outcomes. This article offers insights & practical guidance for data & ML engineers seeking to improve their machine learning workflows with streaming their capabilities, illustrating how the amalgamation of streaming data processing, feature engineering & cloud-native principles can enable the creation of sophisticated, actual time applications.*

Keywords - *Streaming Feature Store, Real-Time Machine Learning, Online Inference, Cloud-Native, Kubernetes, Kafka, Flink, Feature Engineering, MLOps, Low-Latency ML, Serverless ML.*

1. Introduction

1.1. Background and Motivation

A huge change has happened in the design and use of data-driven systems in the last several years. In the past, machine learning (ML) pipelines commonly used batch processing. But nowadays, systems need to be able to adapt to new input in real time, sometimes in milliseconds. The growing requirement for actual time decision-making in areas like fraud detection, personalized suggestions, dynamic pricing, logistics & cybersecurity has driven this transformation.

For example, think about how to find fraud in the financial services industry. A payment that takes a few minutes to verify can come too late to stop a fraudulent transaction. Recommendation systems for e-commerce or media streaming sites must also quickly adapt to changes in user behavior, such as recent clicks or views, in order to suggest the best next purchase or movie. In many other cases, even a few seconds of delay might mean less money or a poorer customer experience.

A fundamental challenge in ML that is key to this actual time transformation is the novelty of features. To correctly show the most recent state of things, features the traits or signals that ML models use to make these predictions must be up to date. In batch systems, features are computed on a regular basis, such every few hours or every day. This method worked well for a number of past uses, but it is no longer good enough for making decisions in real time.

As a result, a number of companies are utilizing their streaming feature stores, which are systems that are meant to constantly compute, store & offer features as the latest information comes in. These stores make sure that the latest features are always available and make it easy for the actual time inference layer to use them with little delay. As more and more people use cloud-native designs, this tendency becomes stronger. Cloud-native designs make it easier to scale, make systems more reliable, and manage resources better in different places.

1.2. Problem Statement

It's clear that we need machine learning systems that work in real time, but putting them into practice is exceedingly hard.

Traditional feature stores, which are usually built on their batch ETL procedures and data warehouses, weren't meant for actual time, low-latency access. Batch processing causes delays, which leads to features that are out of current & differences between the training and serving settings. Also, these older systems often require a lot of human help

and extra logic, which makes it more likely that data will drift and model performance will drop.

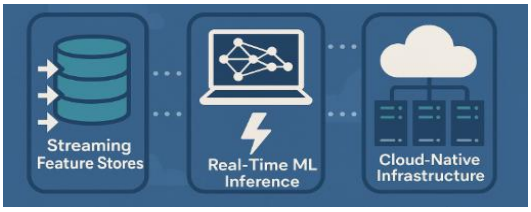

Fig 1: Architecture of Streaming Feature Stores for Real-Time ML Inference

Consistency is a big issue with real-time apps. The attributes used for offline model training must align with those available during inference. It is quite hard to maintain their parity in distributed systems, especially across microservices or many other data pipelines. Any difference, no matter how little, may have a big effect on these model predictions.

Another big problem is latency. When setting up models for actual time use, speed of inference should be a top priority. A model that is well-trained and very accurate is useless if it takes a long time to make predictions. End-to-end latency, which includes everything from taking in data to translating features to making predictions, must be very low, often less than 100 milliseconds.

Scalability is also very important. The system has to be able to handle a lot of information and user interactions without slowing down. Cloud-native systems provide auto-scaling, container orchestration & event-driven execution. However, building durable streaming pipelines on these platforms demands comprehensive expertise & precise architectural planning.

In short, businesses are facing three problems:

- Ways to keep new features coming in distributed systems
- Ways to make sure that training and inference are the same
- Ways to use cloud-native technologies to provide low-latency inference at scale

These challenges show that we need new machine learning infrastructure that can natively handle streaming and real-time apps.

1.3. Objectives

This article aims to bridge the gap between the theoretical promise and practical implementation of real-time machine learning systems. It looks at the concept and structure of streaming feature stores and how they make it possible for real-time inference in cloud-native applications.

We will begin by examining the architecture of these modern feature stores, focusing on the shift from batch processing to streaming-first methodologies. We will look at how streaming data systems (like Apache Kafka or Pulsar), online databases (like Redis, Cassandra, or DynamoDB), and processing frameworks (like Apache Flink or Spark Structured Streaming) work together to make actual time pipelines possible.

Next, we will look at the inference part of the equation, which includes model serving infrastructure (like TensorFlow Serving, KFServing, or Triton Inference Server), online prediction APIs, and how they work with feature stores to run quickly.

The focus will be on cloud-native deployment methods. We will look at Kubernetes, serverless computing & event-driven services, which are the building blocks of scalable, reliable ML pipelines in production. The basic principles & patterns are usually the same no matter which cloud service you use, whether it's AWS, GCP, Azure, or an open-source stack.

To ground the subject in realism, we will provide an extensive case study illustrating the deployment of a real-time recommendation system using a streaming feature store and a cloud-native machine learning inference architecture. The goal is to not just say what is possible, but also show how to make it happen.

By the end of this article, readers should have a full understanding of:

- The importance of streaming feature storage in modern machine learning systems
- Ways to plan and carry out real-time inference procedures
- What tools, patterns, and structures help people do well in cloud-native environments?

Our goal is to provide ML engineers, data architects, and DevOps teams the strategic insights and practical advice they need to build the next generation of smart, responsive products.

2. Evolution of Feature Stores

2.1. What is a Feature Store?

A feature store is basically a single place to store, organize, and send machine learning (ML) features. It plays an important role in the ML lifecycle by acting as a bridge between raw data and machine learning models.

Features are the input signals that ML models use to make these predictions. For example, a user's recent purchases, their location, or how long it has been since they last logged in. In the past, data scientists generated these traits on an as-needed basis and kept them in separate places, such as notebooks, scripts, or any other databases. This led to duplication, inconsistencies, and a lack of supervision.

To fix this problem, feature shops were set up. They collect and show features in a systematic way, which lets teams use the same feature definitions on several projects. There are two main groups.

- Offline feature stores: Used to train models. They get historical information, combine it via batch processing, and then put it in data lakes or warehouses.
- Online feature stores: Used for making decisions in actual time. They let you quickly get to the latest features with little delay, usually by employing in-memory or key-value storage systems.

The best thing about a feature shop is that you can use it again and again. If a feature works well in one model, it may be used in other models without having to rewrite the code. This saves time and makes sure that training and inference are the same. Also, having one source of truth makes it easier to manage, regulate versions, and audit data pipelines.

2.2. Limitations of Traditional Feature Repositories

Feature stores have organized machine learning processes, although most of the ones that came before them were made using batch-first architectures. This design choice creates big problems for actual world applications that need to be done quickly, including fraud detection, recommendation systems, or predictive maintenance.

The first problem is latency. Conventional systems usually get data via nightly or hourly batch procedures. Because of this delay, models use previous information to make predictions, which is not acceptable when trying to find credit card fraud in real time.

Staleness gets in. After a characteristic is developed, stored, and retrieved, its value could not precisely reflect the current state of things. This obsolescence might make your model less effective, especially in fast-moving fields like stock trading or online advertising.

There is also the worry of duplication. In certain systems, one team may use Spark to build training features and store them in a warehouse. At the same time, another team may use Redis or a NoSQL database to build a completely different pipeline that makes those features available online. This duplication typically leads to training-serving skew, which means that models work differently in production than they do during development.

In short, traditional feature stores, even if they are powerful, were not meant to handle real-time data needs. Their design reflects the batch-oriented context from which they came.

2.3. The Beginning of Streaming Feature Stores

As actual time machine learning applications become more common, a new kind of streaming feature storage has emerged. From the start, these systems are designed to handle the constant flow of information, changes, and delivery, all in a matter of seconds or even milliseconds.

We can't wait for the next batch execution in modern apps. Think about ride-hailing apps. The system has to use dynamic pricing tactics that change based on traffic, driver availability, and current demand. This data changes every second. This means that features need to be made and found in less than a second.

Streaming feature stores alleviate this problem by linking to stream processing engines like Apache Kafka, Flink, or Spark Structured Streaming. Instead of waiting for data to build up, they process it as soon as it arrives, making changes, combining data, and saving the outcomes in online repositories so that models may utilize them right away.

This change in architecture provides a number of benefits:

- Freshness right away: Features show the most up-to-date information, which makes models more accurate as things change.
- Unified pipelines: The same logic may be used for both training & inference, which reduces differences between training and serving and makes deployment easier.
- Low-latency inference: By combining streaming ingestion with fast key-value databases like Redis and DynamoDB, feature values may be accessible in milliseconds during prediction intervals.

In the end, streaming feature stores are the next step in the evolution of machine learning operational maturity. They fit with the general trend towards cloud-native and event-driven architectures, which let ML systems adapt to real-time situations instead of only reacting after an occurrence.

As we use AI more and more in different decision-making processes, from logistics to customisation, the ability to work in real time will go from being a nice-to-have to a must-have. Streaming feature stores will be very important in making that happen.

3. Cloud-Native Infrastructure for Real-Time ML

The move to actual time machine learning (ML) has led to improvements in the infrastructure. Traditional monolithic systems do not meet the needs for high-speed data, low-latency inference & continuous deployment. Cloud-native architecture is a modern way of harnessing the cloud's flexibility & the scalability to make it easier to stream data and make actual time inferences. This section looks at the basic parts of these cloud-native systems, the many other tools that make them work, and how they all work together to create strong actual time ML pipelines.

3.1. Principles of Cloud-Native Architectures

Containerization, orchestration, microservices, and declarative APIs are the main principles of cloud-native infrastructure. All of these parts work together to create these systems that are flexible, strong & scalable, and that can quickly adapt to the needs of actual time machine learning processes.

3.1.1. Putting Things in Containers

Containers, like Docker, let developers put a programme & all of its dependencies into a single, portable unit. In

machine learning, this means that you can put models, feature processing algorithms, or serving layers into settings that can be used again and over again. This makes the "it worked on my machine" problem a lot less of a problem and makes it easier to move parts across environments, such as local, staging, or production.

3.1.2. Working together

Containers are powerful, but managing hundreds or thousands of them is impossible without orchestration. At this stage, Kubernetes steps in. Kubernetes handles deployment, scaling, health checks, and failover automatically. It makes sure that your ML services (such feature transformers and inference APIs) are always available and responsive, even when there is a lot of traffic or a node goes down.

Microservices Cloud-native machine learning systems usually don't use these monolithic structures. They split up functionality into smaller services. One service may take in streaming data, another can change features & a third can send out forecasts. Each part may grow on its own, be made in private & be put into use without affecting the rest of the system.

3.1.3. Application Programming Interfaces That Are Declarative

Lastly, declarative APIs and infrastructure-as-code (IaC) methods let teams use code to describe the desired state of the system (for example, Kubernetes YAML manifests or Helm charts). This makes it easier to maintain their versions, keep environments consistent & deploy consistently, all of which are important in the MLOps operations.

3.2. The Environment for Tools

To build real-time machine learning systems in a cloud-native way, you need to connect tools at several levels, such as computer, communications, feature storage, and inference runtimes. Let's look at some of the most important players in the ecosystem.

3.2.1. The Infrastructure and Platform Layer

- Kubernetes: The standard way to manage containerized apps. Most cloud-native ML infrastructures are built on top of it.
- Knative is built on Kubernetes and makes it easier to deploy these serverless applications. This is especially helpful for model inference services that need to be able to go down to zero when they're not being used or handle sudden increases in traffic.
- Istio is a service mesh that controls traffic management, load balancing, security & observability between microservices. Istio is very important for regulating rollout techniques (like A/B testing), keeping track of requests & making sure that components can communicate safely with each other in ML pipelines.

3.2.2. Apache Messaging and Streaming Layer

- Kafka is a strong, distributed message broker that powers machine learning algorithms that stream data. It takes care of getting data from sensors, applications, or people in real time.
- Apache Pulsar is a different option from Kafka that makes it easier to use many tenants and replicate data across different locations. Pulsar has been utilized in these situations when better separation between tenants is needed.
- Apache Flink is a powerful engine for processing streams in actual time. Flink may be used in a ML context to clean, change, and combine information in actual time before sending it to a feature store or inference engine.
- Spark Structured Streaming is a good choice instead of Flink since it can handle both batch and micro-batch processing well. Useful in systems that need to combine past and present data.

3.2.3. Feature Store and Serving Layer Redis:

An in-memory data store that is typically used for online feature storage with extremely low latency. Best for providing actual time features during model inference.

- Amazon DynamoDB is a fully managed NoSQL database that can grow without any other problems. It is widely used as a backbone for online feature stores.
- Feast: An open-source feature repository that keeps feature engineering & model training and deployment distinct. It makes it easy to get features in actual time and in batches, utilizing backends like Redis and BigQuery.
- Hopsworks is a big platform that includes a feature store, versioned data storage & tools for deploying models on a wide scale.

All of these tools work together to provide a cloud-native ML stack that makes it easy to stream their information, change it in actual time, and make inferences online.

3.3. Pipelines for Machine Learning in the Cloud

How do these parts work together to make a single pipeline? Let's look at a typical cloud-native setup for machine learning in real time.

3.3.1. Streaming Ingestion

Data is the first step in any process. In a real-time system, data is always being produced by user interactions, sensors, IoT devices, or backend services. Kafka or Pulsar, which are both robust event logs, collect this information.

Every time someone interacts with a website, their clickstream data may be sent to a Kafka topic. A stream processor may then utilize this information.

3.3.2. Changing and Combining Features

When the raw data is taken in, it needs to be turned into important traits. This is when Apache Flink or Spark Structured Streaming comes into play. These frameworks provide transformations like time-windowed aggregates, joined with extra data & statistical calculations.

For instance, Flink may figure out how long a user's average session lasted over the last 30 minutes or find strange activity. You may use an online store (like Redis or DynamoDB) to store these properties so you can access them quickly & at the same time, you can store them in an offline store (like S3 or BigQuery) for training reasons.

Feast makes it possible to centralise feature definitions, which makes sure that everything is the same throughout both training and inference.

3.3.3. REST/gRPC for online serving

The system has to make the characteristics available for inference once they have been computed and stored. A common setup would show a REST or gRPC endpoint on Kubernetes that handles requests from applications that are further up the chain, such a recommendation engine or a fraud detection system.

Let's say a user logs into their account. The application uses Feast's SDK to get the most current features from Redis and send them to the model for inference.

3.3.4. Services for Model Inference

The last step is the actual forecast. Kubernetes manages models as microservices, which are packaged using tools like Docker. You may access these services directly or using Knative to make them work with scale-to-zero.

Model versioning, A/B testing, or shadow deployment, where numerous models run at the same time, are common in these modern deployments. Istio makes it easier to handle complex traffic between services & makes it easier to see what's going on so you can evaluate performance and fix more problems.

Logging, metrics, and tracing are all linked to tools like Prometheus, Grafana, and OpenTelemetry, which make sure that the complete process can be seen and changed.

4. Streaming Feature Store Architecture

Modern machine learning algorithms often rely on actual time insights obtained from data that is always changing. Streaming feature stores provide as a bridge between raw event data & machine learning inference that is fast and more accurate. They keep up with features that are new, consistent, and available in actual time, whether they are being trained or used. This part explains how a streaming feature store is built, focusing on design ideas, basic parts, consistency, and deployment plans. It uses technologies that work well in a cloud-native context.

4.1. Principles of Design

A good streaming feature store architecture has to include a lot of important parts. These basic ideas affect every part of system design and the choices we make about technology.

- Little delay Writing and reading: Latency is very important in actual time machine learning. A fraud detection model on an e-commerce site or a recommendation engine on a streaming service must be able to make decisions in milliseconds. The feature store has to be able to read and write quickly. We use technologies like Redis and DynamoDB because they can read and write information in less than a millisecond, which makes sure that models can always get the most up-to-date data.
- Exactly once Processing: Extra data or missing records might mess up feature values and make model predictions less accurate. A streaming feature store has to make sure that each data point is handled separately. This is particularly important in distributed systems because reprocessing might happen because of retries, network partitions, or consumer failures. When used with strong source semantics, such idempotent producers in Kafka and Kafka Connect, systems like Apache Flink may enforce end-to-end exactly-once guarantees.
- Navigating through time and adding data from the past: Historical data is important for more than simply training; it's also important for debugging and auditing. To get feature values as they were at a specific moment in the past, the feature store must let you do time-travel queries. It also needs the ability to backfill so that late events or changes to the schema may be included to the feature set. People often employ offline storage mediums like Amazon Derr FightersLinked live oramas Parsers contributions lentils to save old data Madrig
- Entity-Centric Retrieval with Time-to-Live Settings: An entity key, such a user ID, transaction ID, or device ID, may frequently be used to access features. The entity to whom each characteristic belongs must define it. Also, not all traits are permanent. Some lose their importance very quickly and need to be thrown away. TTL (Time-to-Live) rules make it easier for old items to be automatically deleted, which helps online databases use less memory.

4.2. Basic Parts

Let's look at the basic parts of a streaming feature store, which all have different jobs in the process of turning raw data into model input right away.

4.2.1. Debezium, Kafka, and Kafka Connect are all part of the ingestion layer.

The ingestion layer is in charge of bringing event data into the system. Apache Kafka is the main message bus that handles event streams with high throughput and fault tolerance. Kafka Connect makes it easier to connect to additional sources, including cloud object storage or relational databases. Debezium is an important part of change data capture (CDC) since it finds changes in source databases and sends them to Kafka topics as event streams. This setup makes sure that all upstream changes have a consistent, append-only data format.

4.2.2. Processing Layer: Flink Tasks for Combining and Changing

The raw data is turned into features that may be used at this point. Apache Flink processes incoming events right away, making changes, joining them, and adding them up. A Flink job may find rolling averages of user activity over a 10-minute period or combine clickstream data with product information. Flink is great for building deterministic and time-sensitive features since it can handle windowing, watermarking, and state management. For online storage, use Redis and DynamoDB; for offline storage, use S3 and Parquet.

The processed features need to be stored in two places:

- Digital marketplace (like Redis or DynamoDB): This has to do with real-time inference. These stores are intended for quick, key-based retrievals and access patterns with little latency.
- The offline store (like S3 with Parquet format) is used to train or re-train models and keeps a complete record of all feature values. Because Parquet files are structured in columns, they are good for big analytical workloads.

4.2.3. Serving Layer: Feast, Custom APIs for Application Programming

After the features are done, the ML models need to be able to use them. The serving layer makes it possible to run queries in real time using entity keys. Feast is a popular open-source feature store that helps keep track of information, schemas, and registries while separating training and serving pipelines. In more advanced setups, custom APIs or gRPC services could provide features that many models or teams can utilize.

4.3. Consistency between Training and Deployment

One of the biggest problems with machine learning systems is keeping the data used to train models and the data that is available during inference the same. When data moves from one step to the next, the model's performance becomes worse, which makes it harder to find and fix bugs.

- Accuracy in Time: This means that while training a model, features must precisely reflect the information that was available at the time of the event. No data exfiltration. This is frequently done by keeping feature values with event timestamps and utilising time-travel queries when joining features. This idea is used by physical stores and technologies like Feast to make features real via point-in-time reasoning.
- Managing Metadata and the Feature Registry: Each feature has its own schema, ownership, and transformation logic written down so that everything stays organised and consistent. Tools like Feast have a built-in feature register that is the only place to find the truth. It also handles versioning, which lets you keep track of lineage in case of schema or transformation changes. This lets you choose whether to reprocess past data or use a new set of features.

4.4. Strategies for Deployment

Cloud-native design makes it easier to deploy and scale feature pipelines by using modular and repeatable parts.

- Use Microservices to Implement Feature Pipelines: You may turn every feature transition into a microservice or a containerised process. With this method, teams may build, test, and use feature pipelines on their own. A pipeline that makes user session features might run as its own Flink task, separate from product metadata features.
- Helm Charts for Deployment: Helm is the package manager for Kubernetes. It makes it easier to set up and run the complete feature store stack. Helm charts are reusable templates that include the settings for Kafka, Flink, Redis, Feast, and monitoring tools. This makes it easy for teams to use the same settings throughout all stages of development, staging, and production.
- Using Prometheus with Horizontal Pod Autoscaler for monitoring and autoscaling: In manufacturing, operational stability is very important. Prometheus gets information from streaming jobs, Kafka brokers, and storage layers. Grafana dashboards provide latency, throughput, and failure rates in real time. The Horizontal Pod Autoscaler (HPA) in Kubernetes uses these measurements to automatically scale Flink tasks or Redis instances up or down in response to higher demand. This lets the system adapt to traffic spikes on its own.

5. Case Study: Real-Time Fraud Detection System

5.1. Problem Context

Detecting fraud in the financial sector is a big concern and is used as a benchmark for actual time machine learning. Finding problems, such a misplaced credit card or suspicious account activity, in milliseconds helps protect customer trust & save financial losses.

Most of the time, traditional fraud detection systems use data that has been processed in batches and rules-based algorithms. In the present day's fast-paced, high-volume environments, such digital payment gateways, banking apps, or cryptocurrency exchanges, even a one-second delay might lead to the acceptance of a fake transaction. The goal is simple but hard to do technically: provide predictions for each incoming transaction within 100 milliseconds.

This case study looks at how a modern, cloud-native architecture makes it possible to find fraud in actual time by combining a streaming feature store, low-latency inference, and a way for deploying that can grow.

5.2. An Overview of the Architecture

The heart of our fraud detection system is a streaming architecture that gathers, changes, and feeds features in actual time. Our makes sure that each prediction uses the most up-to-date data.

5.2.1. Using Kafka to Get Transactions

All transactions that come in are forwarded to a Kafka topic right away. Kafka is the heart of the data pipeline. It processes thousands of events per second from a variety of sources, including as mobile apps, backend services & point-of-sale terminals.

Flink for Streaming Feature Engineering: Apache Flink works with Kafka streams to provide you actual time aggregations and features. Some such examples are:

- Number of transactions that happened in the last two minutes
- Average value of transactions during a 5-minute period
- How many transactions each merchant or location has

These groups help the model understand not just what a transaction is but also how it fits with a user's or card's recent activity.

5.2.2. Using Redis for Feature Provisioning

The engineered features are saved in Redis and have a Time-To-Live (TTL) of 30 minutes. Redis was chosen because it is fast (it can read in less than a millisecond) and it may be used as an online feature store. This means that all the important features are ready to use right away when the model is used for inference, even for jobs that need a lot of processing power.

5.2.3. Using Kfserving And Feast To Deploy Models

The model is distributed via KFServing, which is part of the Kubeflow ecosystem, and is linked to Feast, the feature store. KFServing makes it easy to install high-performance models on Kubernetes. It has autoscaling, versioning, and built-in GPU support as needed. Feast takes care of getting features and makes sure that the features needed for training are the same as those needed for inference. This keeps the difference between training and serving to a minimum.

5.3. Specifications for Execution

Building this system required more than just picking the right technology; it also needed orchestration, automation, and observability from the beginning.

5.3.1. Using GitHub Actions with ArgoCD for Continuous Integration and Continuous Deployment

We use continuous integration and deployment (CI/CD) to make sure that both the model and the data pipeline parts are stable and can be used again and again. GitHub Actions a test automation and Docker image building, and ArgoCD makes sure that Kubernetes manifests are applied declaratively in both staging & production clusters. This setup made it easy to make changes to the Flink tasks, Redis schema, model weights, and Kubernetes deployments quickly and safely.

5.3.2. Architecture that works with Kubernetes

Kafka, Flink, Redis, Feast, and KFServing all work natively on Kubernetes. This had two key benefits: it made everything the same in all settings (local development, staging, and production). Elastic scalability is very useful when there is a lot of traffic, such on Black Friday or payday. Operators for each component, such as Strimzi for Kafka, Flink Operator, Redis Operator, and KFServing controller, take care of the lifecycle, which includes updates, scalability, and failover management.

5.3.3. Versions of Features and Models

To fix problems and make sure things can be done again, version control is very important. Feast's feature registry keeps track of features and gives each transformation pipeline its own version identifier. MLflow also keeps track of model versions and connections to the training data, codebase snapshot, and evaluation metrics.

This openness makes it easier to answer important audit questions, such "Which version of the fraud model was in use at 3:15 PM last Tuesday?" This is something that regulated industries need to do.

5.4. Results and Consequences

The system was checked and tested for latency and how well it could anticipate things.

5.4.1. Time Delay Less Than 100 Milliseconds

The average time it took for an event to be received and a forecast to be sent was less than 100 milliseconds. This includes around 10 milliseconds for Flink processing and Kafka ingestion.

- Retrieving a Redis feature takes around 5 milliseconds.
- Using KFServing, model inference takes around 20 milliseconds.
- Time left over because of network lag and the cost of serializing

This makes sure that actual time fraud checks don't slow down the user experience in any apparent way, whether they are approving a transaction or starting a second verification procedure.

5.4.2. 99.99% Features That Are Available

Redis was incredibly available and had very few bugs. The TTL approach made sure that features stayed up to date without taking up more RAM. Redis' clustering kept feature readings steady throughout failover testing.

5.4.3. Better Accuracy with New Features

The accuracy of fraud detection improved significantly as compared to a baseline batch system. By using up-to-date behavioral information, including recent spikes in transactions, the software was able to find fraudulent conduct faster and with more certainty.

- This led to fewer faulty positives, which made things easier for actual users.
- Faster response to the latest ways of committing fraud
- Users and stakeholders who need to follow the rules will feel more confident.

5.4.4. Strength and Watching

We utilized Prometheus and Grafana to see how the system was doing. Essential dashboards kept an eye on things like Flink task throughput and backpressure.

- Redis cache hit rate
- KFServing's inference delay
- Mistakes in requests for features and alerts for features that are no longer available

The Horizontal Pod Autoscaler (HPA) turned on when there was a lot of traffic and changed the KFServing pods and Redis instances as needed. This automated adaptability kept latency within the set bounds without needing any help from an operator.

6. Conclusion and Future Work

6.1. Summary

At the moment, streaming feature stores are changing how machine learning models work. They close the important gap between training that takes place offline and decision-making that takes place online by adding the newest low-latency features straight to operational systems. These technologies speed up the process of gathering information and drawing conclusions from it. This is important for things like fraud detection, personalization & actual time suggestions.

Cloud-native infrastructure is at the heart of this change. Kubernetes, Apache Kafka, and serverless computing technologies provide you the scalability, reliability & automation you need to handle complex, data-heavy processes. They enable teams to finish tasks more quickly, boost development, and make sure that products are delivered on time without having to worry about the costs of infrastructure.

You need more than simply faster CPUs to enable real-time ML inference. You also need a well-thought-out architecture. Each component, from adding features to making sure that online and offline repositories function together, helps produce predictions that are both quicker and more accurate. When done well, it makes programs smarter and more responsive, so they can alter in real time to suit the needs of users and the environment.

6.2. Future Plans

Looking forward, a number of potential paths are opening up. One is streaming-native feature discovery, which means that new features are learned and built in real time as data is received, rather than being set up ahead of time. This might lead to models that are more flexible & more suited to the scenario.

The combination of large language models (LLMs) with actual time vector search is a current problem. By combining streaming information with LLMs, you can make complex apps like contextual retrieval, semantic alerts & dynamic customization with very little delay.

Upgrades to infrastructure, such as autoscaling GPU-enabled endpoints & using ephemeral computing for serverless inference, might save expenses while keeping things more responsive. Ultimately, improving support for multi-cloud & also edge deployments will make it possible for actual time ML inference to happen in places where latency is important or where information is spread out, as in smart retail or autonomous systems.

References

[1] Gupta, A., & Chaturvedi, Y. (2024). Cloud-native ML: Architecting AI solutions for cloud-first infrastructures. Nanotechnology Perceptions, 20(7), 930-939.

[2] Srigadde, B. R. (2023). The Hidden Gem: Lightning Headless Component. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(1), 244-254. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I1P125

[3] SAMUEL, A. (2021). Cloud-Native AI solutions for predictive maintenance in the energy sector: A security perspective. Available at SSRN 5290068.

[4] Allenki, S. S. (2024). Automating Backups and Recovery: Reducing Manual Work by Over 50%. *International Journal of Emerging Research in Engineering and Technology*, *5*(1), 166-176. https://doi.org/10.63282/3050-922X.IJERET-V5I1P119

[5] Lakarasu, P. (2022). End-to-end Cloud-scale Data Platforms for Real-time AI Insights. Available at SSRN 5267338.

[6] Takkalapally, D. (2023). HoloSearchAI: AI-Driven Latency Optimization Framework for Distributed Search Systems. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(3), 217-227. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I3P122

[7] Shiramalla, R. (2023). Optimizing Cross-Platform Enterprise Integrations Using Workato: A Case Study of Salesforce and Oracle SaaS Applications. *International Journal of Emerging Trends in Computer Science and Information Technology*, *4*(1), 232-243. https://doi.org/10.63282/3050-9246.IJETCSIT-V4I1P124

[8] Joseph, E., & Anderson Jack, A. W. (2024). Hybrid Cloud Architecture for Real-Time Data Streaming Systems.

[9] Kumar Doodala, A. N. (2024). Validating UX consistency Across Omnichannel Platform. *American International Journal of Computer Science and Technology*, *6*(6), 87-97. https://doi.org/10.63282/3117-5481/AIJCST-V6I6P109

[10] Gaddam, R. R. (2021). Hermetic ML Environments using Conda-Lock and Docker. *American International Journal of Computer Science and Technology*, *3*(4), 22-34. https://doi.org/10.63282/3117-5481/AIJCST-V3I4P103

[11] Parasaram, V. K. B. (2022). Converging Intelligence: A Comprehensive Review of AI and Machine Learning Integration Across Cloud-Native Architectures. International Journal of Research & Technology, 10(2), 29-34.

[12] Muppaneni, K. (2022). Optimizing React Hooks for Efficient State and Side-Effect Management. *American International Journal of Computer Science and Technology*, *4*(6), 44-55. https://doi.org/10.63282/3117-5481/AIJCST-V4I6P105

[13] Suryadevara, S. S. K., & Nakirikanti, S. (2024). Blockchain-Backed Content Authenticity Verification Framework. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *5*(1), 242-252. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I1P125

[14] Koneru, S. H., Avireneni, R. T., Reddy Yelkoti, N. K. K. ., & Yerneni Khaga, S. P. . (2021). Cloud-Native Micro services Architecture. *International Journal of Emerging Trends in Computer Science and Information Technology*, *2*(4), 86-94. https://doi.org/10.63282/3050-9246.IJETCSIT-V2I4P110

[15] Vppalapati, M. (2023). When Identity Decisions Throttle Data Movement. *International Journal of Emerging Research in Engineering and Technology*, *4*(3), 160-170. https://doi.org/10.63282/3050-922X.IJERET-V4I3P117

[16] Parakala, A. (2024). Agentic Automation: What's next for Jobs . *American International Journal of Computer Science and Technology*, *6*(6), 25-35. https://doi.org/10.63282/3117-5481/AIJCST-V6I6P103

[17] Rahman, M., Mahbuba, T., Siddiqui, A., & Nowshin, S. (2019). Cloud-native data architectures for machine learning.

[18] Allenki, S. S. (2024). Building Scalable Data Replication Pipelines for Real-Time Analytics. *American International Journal of Computer Science and Technology*, *6*(1), 71-81. https://doi.org/10.63282/3117-5481/AIJCST-V6I1P108

[19] Takkalapally, D., & Takkellapally, M. R. (2023). GC-TuneHFT: AI-Based Garbage Collection Optimization in High-Frequency Trading Environments. *American International Journal of Computer Science and Technology*, *5*(6), 25-37. https://doi.org/10.63282/3117-5481/AIJCST-V5I6P103

[20] Lakarasu, P. (2023). Designing cloud-native ai infrastructure: A framework for high-performance, fault-tolerant, and compliant machine learning pipelines. Fault-Tolerant, and Compliant Machine Learning Pipelines (December 11, 2023).

[21] Shiramalla, R. (2024). Secure Multi-Cloud API Orchestration between Salesforce, Oracle CPQ, and Azure. *American International Journal of Computer Science and Technology*, *6*(3), 102-113. https://doi.org/10.63282/3117-5481/AIJCST-V6I3P108

[22] Sethupathy, A., & Kumar, U. (2020). Cloud-Native Architectures for Real-Time Retail Inventory and Analytics Platforms. International Journal of Novel Research and Development, 5, 339-355.

[23] Muppaneni, R. K. (2022). From Legacy ERP to Cloud-First: A Transformation Story with Dynamics 365. *International Journal of Emerging Research in Engineering and Technology*, *3*(4), 153-164. https://doi.org/10.63282/3050-922X.IJERET-V3I4P117

[24] Nangi, P. R., & Settipi, S. (2023). A Cloud-Native Serverless Architecture for Event-Driven, Low-Latency, and AI-Enabled Distributed Systems. *International Journal of Emerging Research in Engineering and Technology*, *4*(4), 128-136. https://doi.org/10.63282/3050-922X.IJERET-V4I4P113

[25] Suryadevara, S. S. K. (2024). Resilient Multi-CDN Delivery Model Using AI-Based Traffic Switching for Global AEM Deployments. *International Journal of Emerging Trends in Computer Science and Information Technology*, *5*(3), 191-200. https://doi.org/10.63282/3050-9246.IJETCSIT-V5I3P119

[26] Kumar Doodala, A. N. (2024). Service Virtualization for API-First development: A shift-Left Testing Strategy. *American International Journal of Computer Science and Technology*, *6*(4), 50-58. https://doi.org/10.63282/3117-5481/AIJCST-V6I4P105

[27] Xiong, J., & Chen, H. (2020, November). Challenges for building a cloud native scalable and trustable multi-tenant AIoT platform. In Proceedings of the 39th international conference on computer-aided design (pp. 1-8).

[28] Parakala, A. (2024). Self-Learning Bots & Cloud-Native Platforms. *International Journal of Emerging Trends in Computer Science and Information Technology*, *5*(4), 132-141. https://doi.org/10.63282/3050-9246.IJETCSIT-V5I4P114

[29] Gaddam, R. R. (2021). Vertex AI as a Unified Control Plane for MLOps. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *2*(2), 92-102. https://doi.org/10.63282/3050-9262.IJAIDSML-V2I2P110

[30] Visser, R. (2024). Cloud-Native AI and Data Governance Architectures for Real-Time Fraud Detection and Financial Risk Prediction. International Journal of Technology, Management and Humanities, 10(03), 141-149.

[31] Muppaneni, K. (2022). Comparative Analysis of Client-Side Storage Mechanisms. *International Journal of AI, BigData, Computational and Management Studies*, *3*(1), 171-182. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I1P119

[32] Qaman, A. A., Beber, T. R., & Allen, Q. M. (2023). AI-Driven Network Traffic Anomaly Detection Using Contrastive Learning in Cloud-Native Infrastructures. Journal of Advanced Artificial Intelligence Research, 2(1).

[33] Muppaneni, R. K. (2022). Data Privacy in the Age of AI: How Dynamics 365 Handles Regulatory Challenges. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *3*(4), 159-170. https://doi.org/10.63282/3050-9262.IJAIDSML-V3I4P117

[34] Srigadde, B. R. (2023). Creating Object Quick Actions with Lightning Web Components. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(2), 167-180. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I2P118

[35] Mamun, A., & Saidur, M. J. I. (2021). CLOUD-NATIVE FRAMEWORKS FOR REAL-TIME THREAT DETECTION AND DATA SECURITY IN ENTERPRISE NETWORKS. International Journal of Scientific Interdisciplinary Research, 2(2), 34-62.

[36] Vppalapati, M., & Talasila, P. K. . (2023). Unobservable Performance: Storage Failures That Leave No Metrics Behind. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, *4*(4), 177-188. https://doi.org/10.63282/3050-9262.IJAIDSML-V4I4P120

[37] Zahra, F. T., Bostanci, Y. S., Tokgozlu, O., Turkoglu, M., & Soyturk, M. (2024). Big Data Streaming and Data Analytics Infrastructure for Efficient AI-Based Processing. In Recent Advances in Microelectronics Reliability: Contributions from the European ECSEL JU project iRel40 (pp. 213-249). Cham: Springer International Publishing.

[38] Veershetty, G. (2025, June 11). Designing clean-core extension architectures for RISE with SAP using SAP BTP: A reference model and evaluation framework. SSRN. https://doi.org/10.2139/ssrn.6749501

[39] Taluri, R. (2024). A Cloud-Native Reference Architecture for Data Engineering, Generative AI, and Decision Intelligence Using AWS and Amazon Bedrock. International Journal of AI, BigData, Computational and Management Studies, 5(1), 218-227. https://doi.org/10.63282/3050-9416.IJAIBDCMS-V5I1P122