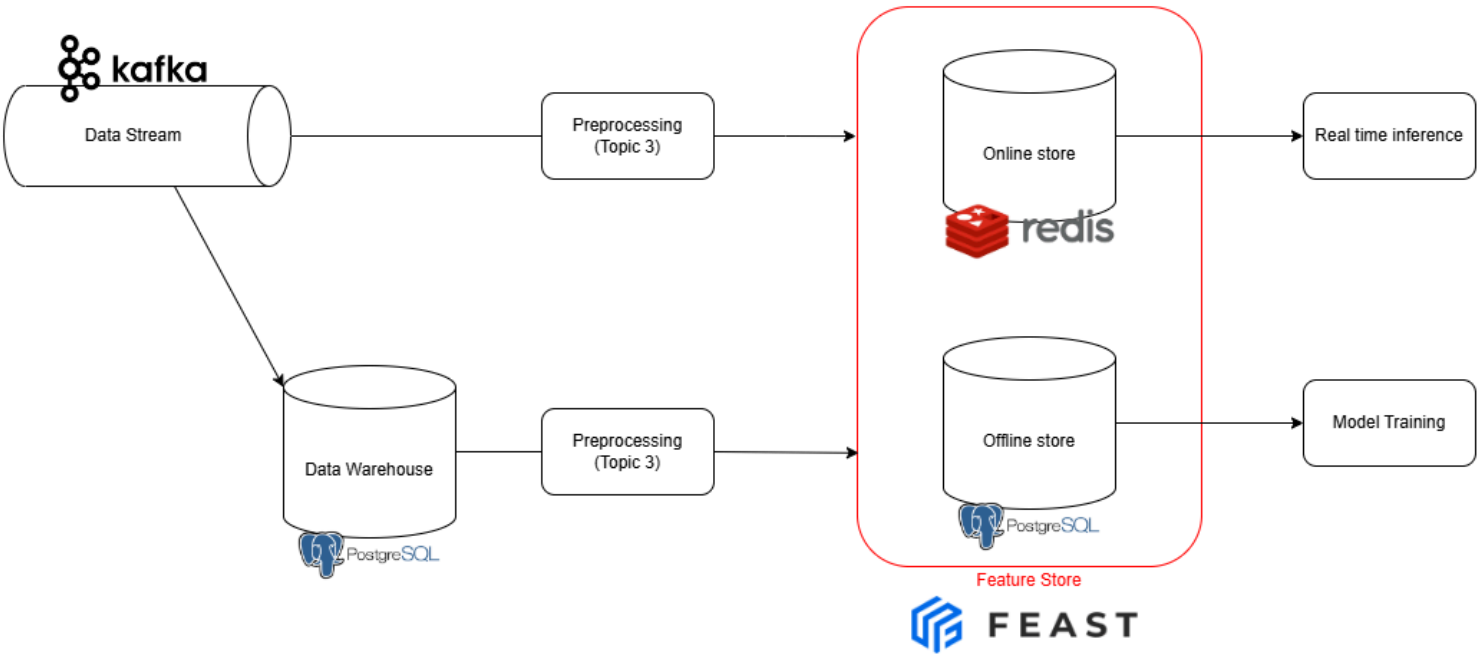
Milestone 1

Architecture



Our system is designed to handle both batch data stored in the Data Lake and real-time streaming data from various sources. This flexibility allows us to process both historical and real-time data, enabling us to meet different operational requirements.

Data Ingestion

- *Batch Data:* Historical data is stored in a PostgreSQL Data Lake, which serves as the central repository for bulk data storage. This data can be loaded for batch preprocessing when required.
- *Real-Time Data:* Streaming data is ingested through a Kafka stream, allowing us to capture data in real-time. For this data, we have two options:
    - It can be directly sent to the Data Lake, where it is stored and then preprocessed in batches.
    - Alternatively, it can undergo immediate, real-time preprocessing before it moves to the Feature Store.

Preprocessing

Both batch and real-time data are processed through a series of transformations to prepare it for feature storage. Preprocessing can occur directly on incoming data streams or on data retrieved from the Data Lake. This ensures that features are consistently formatted and processed, regardless of the data source, before they are sent to the Feature Store.

Feature Store Implementation with Feast
Our Feature Store, managed by Feast, is a dual-layered storage solution that meets both real-time and historical data requirements:

- **Online Store** (SQLite): This layer serves low-latency data to support real-time inference needs. By using a high-speed, lightweight database, the online store provides quick access to the latest feature values, enabling fast responses for real-time applications. This is essential for use cases where latency needs to be minimized, such as providing insights or KPI updates to teams monitoring ongoing operations.
- **Offline Store** (PostgreSQL): The offline store provides a comprehensive historical record of features, stored over time. This allows us to access a rich set of historical data to train and retrain machine learning models. By having access to a large volume of historical data, our models can be trained with high-quality, well-defined features, improving their accuracy and robustness.

Feast's Role in Feature Management
Feast (Feature Store for Machine Learning) is a critical tool in our architecture, simplifying and automating feature storage, access, and retrieval. It abstracts the complexity of managing feature consistency across online and offline stores, ensuring that:

- Features used for real-time inference are identical to those used for training, which helps prevent data leakage and feature drift.
- Data consistency is maintained across different environments, improving the reliability of our models in production.
- Features are efficiently retrieved for both inference and training, streamlining data flow and reducing the engineering effort required for feature engineering and maintenance.

Application of Feature Store for Model Training and Inference

- *Model Training:* The offline store is queried during model training to retrieve historical features. This data is crucial for creating accurate machine learning models that can generalize well on unseen data.
- *Real-Time Inference:* The online store allows us to quickly retrieve the latest feature values during real-time inference. This is essential for scenarios requiring immediate insights, such as real-time monitoring and dynamic KPI calculation.

Feature store comparison

Feature Store	Cost	Scalability	Ease of Integration	Data Consistency (Batch & Real-Time)	Community/Support
Feast	Free (open-source)	Highly scalable, especially with Kubernetes	Easy integration with Kafka, Spark, cloud services	Ensures consistency across stores	Strong open-source community, backed by Tecton and Google
Tecton	High (commercial license)	Enterprise-level scalability, fully managed	Easy integration with ML workflows	High consistency, automated monitoring	Enterprise support, rapid updates
AWS SageMaker Feature Store	Variable, based on AWS usage fees	Scales well within AWS ecosystem	Excellent within AWS, limited with other platforms	High consistency, AWS-integrated	Fully managed by AWS, good support
Hopsworks	Free (open-source), paid options for advanced features	Scalable, but complex setup	Integrates well with Spark, TensorFlow	Strong consistency, data versioning	Growing open-source community
Custom Solution	High (development and maintenance costs)	Customizable to specific scalability needs	Depends on internal resources and infrastructure	Variable, depends on implementation	Limited to internal resources

We chose Feast as our feature store because it offers a balance of **cost-effectiveness**, **scalability**, and **ease of integration** that meets our project requirements.
As an open-source solution, Feast minimizes licensing costs while still providing the flexibility to scale with our data needs. Its **compatibility** with both **real-time and batch data** ensures consistency across training and inference, reducing risks of feature drift.
Additionally, Feast integrates smoothly with popular tools like **Kafka** and **Spark**, making it straightforward to implement within our existing infrastructure. For these reasons, Feast stands out as the optimal choice, delivering robust feature management without the high costs of commercial solutions.

Online Store Comparison

Online Store	Latency	Cost	Ease of Use	Scalability	Data Persistence	Integration with Feast
SQLite	Low (good for small, single-node setups)	Very low (open-source, minimal resource usage)	Very easy (lightweight, embedded DB)	Limited natively, but can scale with Kubernetes (with complexity)	Persistent on disk, but not fault-tolerant	**Supported** (lightweight option for low-throughput use cases)
Redis	Extremely low (optimized for in-memory data)	Moderate (open-source, but higher memory costs)	Moderate (requires some setup, needs memory management)	High (can be distributed and clustered)	Volatile (persistent options exist but less reliable for critical data)	**Supported** (optimized for high-speed feature retrieval)
Amazon DynamoDB	Low (optimized for AWS environment)	High (usage-based, can get costly at scale)	Easy within AWS, managed service	Very high (fully managed, automatically scales)	Persistent, highly available	**Supported** (well-suited for scalable, managed solutions in AWS)
Cassandra	Low to moderate (depends on setup)	Low to moderate (open-source, but requires infrastructure)	Moderate to complex (requires tuning, setup)	Very high (distributed, scales horizontally)	Highly persistent and fault-tolerant	**Limited** (requires custom setup; less common in Feast deployments)

When choosing the best online store, we prioritized **latency, scalability, ease of integration with Feast, and cost-effectiveness**. Here's how each option compares:

- **Amazon DynamoDB** offers excellent scalability, low latency, and easy integration within the AWS ecosystem. However, its usage-based cost model can be expensive, which doesn't fit well with our cost-control requirements.
- **Cassandra** is highly scalable and fault-tolerant, making it suitable for distributed, high-availability applications. However, it has limited native integration with Feast, requires custom configuration, and can be complex to set up.
- **SQLite** is cost-effective, lightweight, and easy to use, with native support in Feast. While it lacks native scalability, it can scale with Kubernetes, but this adds complexity and may not be as efficient for high-throughput use cases.

After evaluating these options, **Redis emerges as our top choice**. It offers extremely low latency, excellent scalability through clustering, and is well-supported by Feast for high-speed feature retrieval. Redis's in-memory design and Feast compatibility make it a strong fit for our architecture, allowing for responsive, real-time performance with efficient scaling potential.

Thus, we chose Redis as our online store for its ideal balance of **latency, scalability, and integration**, ensuring both high performance and adaptability in our feature store architecture.

Data Warehouse Comparison (also valid for offline store)

Platform	Cost	Ease of Use	Scalability	Management	Privacy
Amazon Redshift	Relatively high pricing due to constant computation needs.	Straightforward for AWS users; setup may require additional work for data transfer.	Scales well for petabyte-level data but requires manual adjustments.	Requires significant management, especially in high-computation environments.	Strong privacy and security features, with AWS compliance.
BigQuery	Cost-effective for periodic large queries, billed by data processed.	Easy learning curve, especially for GCP users.	Easily scalable, ideal for ad-hoc or occasional workloads.	Highly automated, users only manage datasets and queries.	Robust security integrated with GCP's privacy features.
Snowflake	Flexible pricing, pay separately for storage and compute.	Quick to set up, supports multiple cloud platforms.	Seamless scaling for concurrent users and continuous usage.	Simplified with automated scaling and separation of storage/compute.	Dependent on provider, generally very secure
PostgreSQL	Free open-source; operational costs increase with scaling.	Simple for teams with experience, open-source for local/cloud deployment.	Limited scalability (up to ~1TB); performance may degrade beyond that.	Requires active management for large datasets; limited scalability.	Limited security features but configurable privacy settings.

We have chosen PostgreSQL primarily for its **open-source** nature, which allows us to have greater control over deployment and configuration without the need for proprietary licenses. While other options like Amazon Redshift, BigQuery, and Snowflake offer **more convenient management**, **better scalability**, and **robust cloud-native features**, they come with higher operational costs or require specific cloud ecosystems. For our use case, PostgreSQL

provides a **cost-effective solution** for handling smaller datasets and allows us to **manage privacy settings** according to our requirements, though its scalability limits might pose challenges as our data grows.

How to Address Scalability for Data Warehouse and Offline Store

The choice of PostgreSQL is valid both for the data warehouse and for the offline store of the feature store. However, scalability issues are more pronounced in the context of the **data warehouse**, where large datasets can cause performance degradation, especially as data grows beyond ~1TB. In contrast, the offline store of the feature store typically handles smaller datasets, so scalability is less of a concern there.

To address the scalability limitations for the data warehouse, one potential solution is to migrate to a more scalable platform like **BigQuery**, which can efficiently handle larger datasets and scale as needed. Migration from PostgreSQL to BigQuery is relatively straightforward, given the compatibility of SQL syntax and the tools available for data transfer.

Alternatively, PostgreSQL can be scaled using **Google Cloud's managed PostgreSQL clusters**. These clusters provide more resources than a local PostgreSQL instance and can handle larger datasets. However, this solution still requires paying for the cloud infrastructure, which can become costly depending on the scale and usage. Google Cloud offers various options for scaling PostgreSQL clusters (e.g., through vertical scaling or adding more nodes), which might help in alleviating some of the scalability issues, but it comes at a price.

Data Model

Data Warehouse
In the Data Warehouse, to effectively store measurements from various sensors over time, our system must address the following requirements:

1. Efficient management of time-series data.
2. Support for multiple sensor types.

Moreover, the system must be adaptable, enabling seamless integration of new sensor types without extensive reconfiguration. This flexibility suggests that a standard SQL schema, while computationally efficient and fast, is too restrictive to accommodate evolving sensor needs.

To balance performance and adaptability, we propose a hybrid approach that combines SQL tables with a JSON-based data format. This design allows the system to ingest data from different sources without requiring structural changes, ensuring easy scalability and broad compatibility. By implementing this approach, we can serve a wide range of customers with different requirements and adapt to new data types over time as customers change their sensor configurations or express interest in additional data collection.

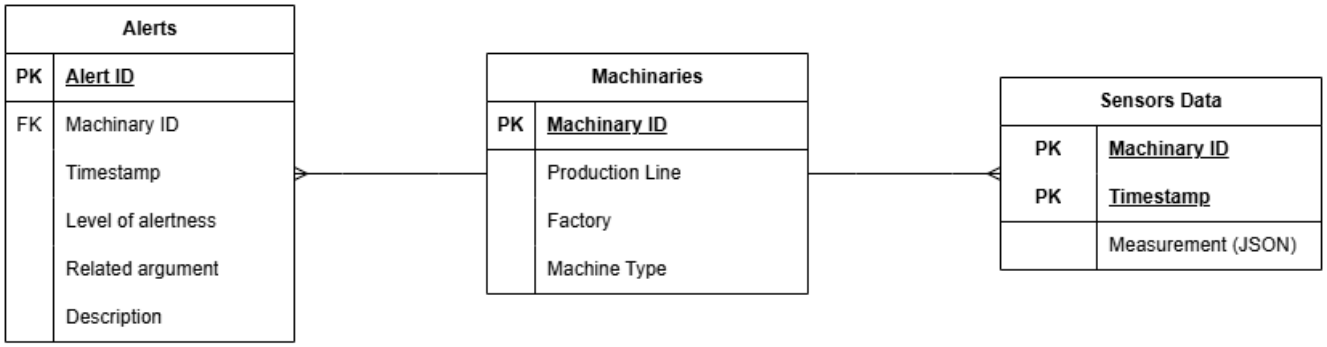
Feature Database

The feature database serves a distinct role from the data warehouse, focusing primarily on the storage of processed features rather than raw sensor measurements. Unlike the data warehouse, where flexibility is essential to accommodate new and evolving sensor types, the feature database emphasizes stability and structure. Here, we store only the specific, engineered features required for model training, rather than raw machine measurements. Consequently, the feature database is not subject to frequent schema changes based on sensor updates, allowing for a more consistent, tabular storage format optimized for machine learning applications.

In this database, data is organized in a highly structured, tabular format that aligns with the needs of machine learning workflows. Tabular data is well-suited for model training, as it allows for straightforward access to features and efficient handling of large datasets. The structured nature of this data ensures compatibility with various machine learning algorithms, enabling them to process and learn from the stored features effectively.

This focused, structured approach to data storage in the feature database is ideal for maintaining high-performance data access and enabling effective model training, while also ensuring that features remain consistent and up-to-date across various machine learning applications.

Data Schema



Apache Kafka an event streaming platform

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Kafka combines three key capabilities:

1. To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems.
2. To **store** streams of events durably and reliably for as long as you want.
3. To **process** streams of events as they occur or retrospectively.

Kafka consists of **servers** (which store and distribute data) and **clients** (which send and read data for monitoring or analysis).

In an example of the machinery data in a company, Kafka acts as an intermediary system between machines that generate streaming data and applications that use this data for monitoring and analysis . Here's how Kafka is set up in this context.

Kafka Components and Functionality

1. **Kafka Servers (Brokers and Kafka Connect)**:
    - **Brokers** are Kafka servers that store and distribute data. Data from the machinery (such as temperature, speed, and energy consumption) is organized into **topics** by category. Brokers ensure that data is replicated across multiple servers, providing reliability and operational continuity.
    - **Kafka Connect** is a component that makes it easy to integrate Kafka with other systems. For example, it can export data collected by Kafka to a data warehouse for long-term storage or historical analysis, and it can also import data from other sources into Kafka.
2. **Kafka Clients (Producers and Consumers)**:
    - **Producers** are applications or devices that collect data from machinery and send it to Kafka. Each machine could have software or an IoT device that sends real-time specific data to Kafka, like temperature or operational speed, by publishing it to the appropriate topic.
    - **Consumers** are applications that read and process data from Kafka topics. In our case it can include: preprocessing data (before being inserted into feature store) or data warehouse storage.

Kafka topics

Kafka stores **events** (like records or messages) with a key, value, and timestamp. **Producers** write events to **topics**, and **consumers** read them. Topics are like folders that can have multiple producers and consumers. Events in Kafka are not deleted after consumption but are stored for a configurable time.

Topics are **partitioned** across multiple Kafka brokers to ensure scalability. Events with the same key (e.g., customer ID) are grouped in the same partition, and Kafka guarantees that events in a partition are read in the order they were written. This architecture ensures high performance and scalability.
To make your data fault-tolerant and highly-available, every topic can be **replicated**

Why Kafka

Table of other possible technologies:

Technology	Pros	Cons
Apache Pulsar	- **Multi-tenancy** and **dynamic scalability**. - **Low latency** for real-time use.	- Kafka has a more mature ecosystem and is easier to integrate
RabbitMQ	- **Simple to set up**. - **Low latency** for message queues. - Supports multiple protocols	- Not suited for high message volumes (doesn't support message batching) - Limited durability.
Amazon Kinesis (AWS)	- **Managed** with **auto-scaling**. - **AWS integration**.	- **Paid service** with variable costs. - AWS-dependent.
Google Cloud Pub/Sub	- **Fully managed**. - Integrated with Google Cloud. - **High scalability**	- **Paid service**. - Google-dependent.

Unlike **Amazon Kinesis** and **Google Cloud Pub/Sub**, which are paid services and tied to specific cloud platforms, **Kafka** (which is free) offers greater flexibility and independence. Additionally, compared to **Apache Pulsar** and **RabbitMQ**, Kafka has a larger community, a wider range of tools, and greater maturity, making it a more versatile choice for real-time data streaming needs.