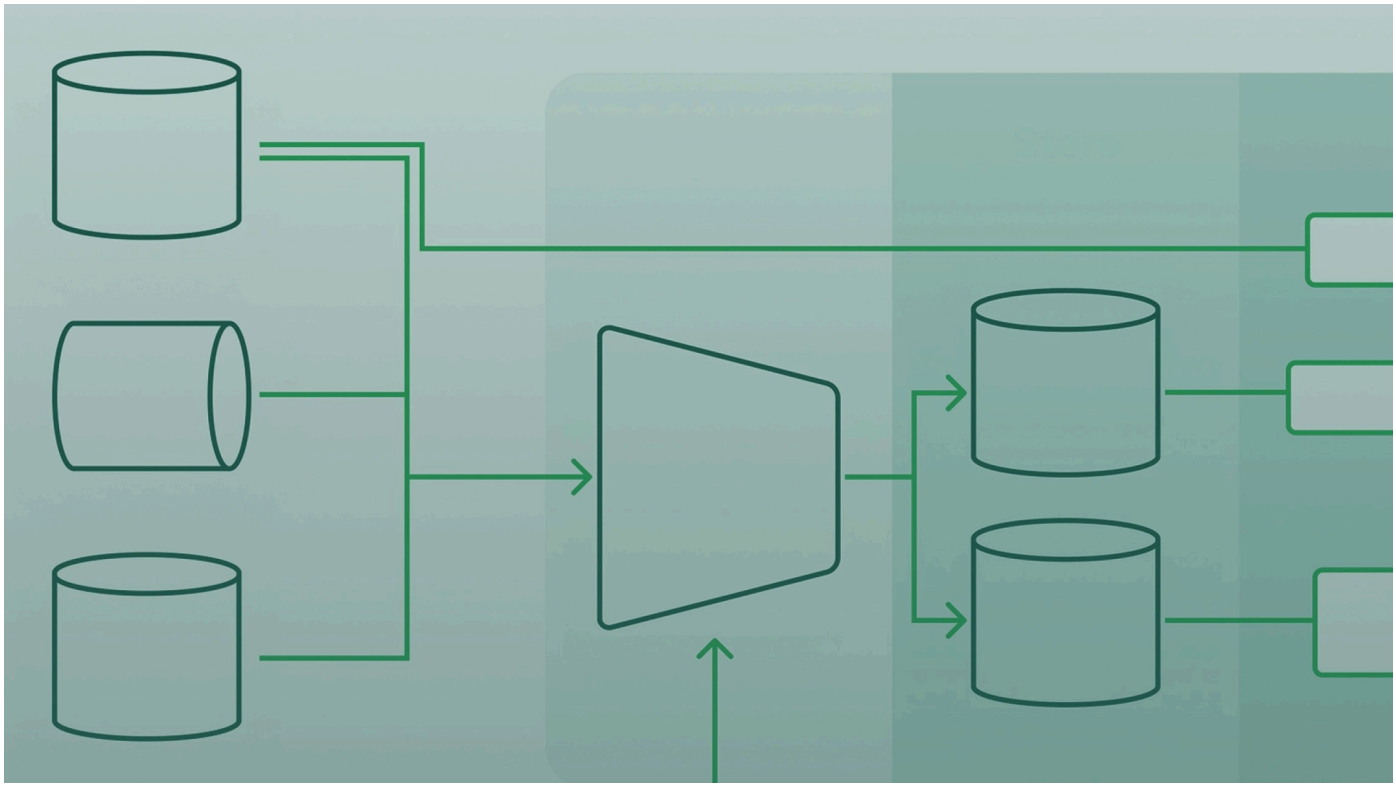

What is a Feature Store?

An Architectural Deep Dive

June 4, 2024


by Melanie Chen, Samuel Mignot

As more and more companies implement machine learning, especially for real-time applications such as detecting fraudulent credit card swipes or making critical diagnostic decisions, the best ML teams rely on tooling that enables engineering excellence and collaboration.

These applications are built not only on data, but on data transformations. Machine learning models are the most concrete example of this: they take transformed data (engineered features) and further "transform" them into outputs—predictions, embeddings, etc. However, data transformation extends beyond modeling. Other common examples include:

- combining features from different data sources (for instance, events from a stream, rows from a relational database, and responses from API requests)
- performing complex windowed computations or aggregations.

Making transformed features production-grade—that is, making them consistent, debuggable, monitorable, and fresh—is a hard problem. However, failing to address it causes problems including but not limited to:

- Slow queries and model deployment cycles
- Inefficient recomputation of expensive features
- Unmonitored feature drift, divergence in feature definitions, and model performance degradation
- Lack of transparency in feature lineage, dependencies, and evolutions, required for iteration and data governance
- High storage overhead

Feature Stores address these challenges by:

1. centralizing and remembering every feature you've ever computed
2. allowing features to be queried in real time, and
3. allowing historical data to be queried, providing snapshot views into your data.

> For a detailed case study on how a feature store can be used to build a fraud detection system, refer to Feature Store at Work: A Tutorial on Fraud and Risk

In the next couple sections, we take a closer look at the subcomponents of a feature store, which will illustrate how feature stores work.

The Basic Architecture of a Feature Store

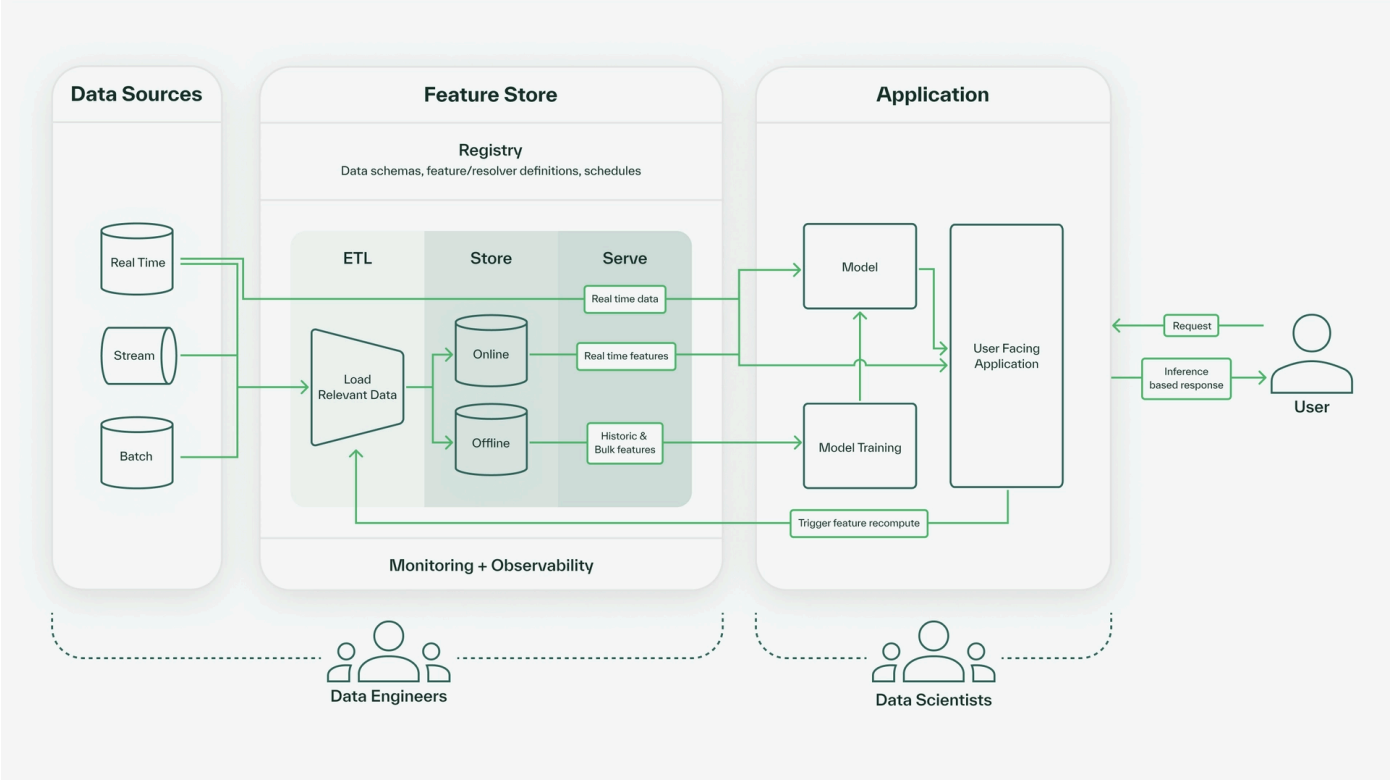

Diagram of the basic layout of a feature store

Feature stores are defined by four core components in the flow of data:

- **Sourcing**: connecting to raw data sources,
- **Transforming**: loading and running transformations on data,
- **Storing**: persisting transformed features, and
- **Serving**: providing access to transformed data.

There are two additional (though equally important) components that extend naturally from the core components and respond to the challenges that feature

stores address:

- **Monitoring**: detecting feature drift, shifts in latency and storage usage, and data freshness.
- **Experimenting**: iterating on feature and resolver definitions during model training.

These two components aren't definitional, but they are essential parts of making transformed features production-grade.

Sourcing

Feature stores are built to be data source agnostic. They implement connectors that allow data to flow in from your company's upstream data sources, making them easy to integrate into any existing data architecture. Common categories of data sources include real-time, streaming, and batch data sources from which the feature store can load feature data and all the input data required to generate feature data.

Transforming

Feature stores compute and persist transformed data. Users write most of their custom logic and code for the transformation step of a feature store. The goal of transformation is to specify the desired structure of your **features** and the logic with which to compute feature values. These specifications comprise the **registry** for the feature store, which serves as a schema for all current and historical feature definitions. The registry should contain definitions for the input data sources or features and specific computations required to determine a feature value.

Once the registry has been defined and deployed, then feature stores further optimize performance by performing these data transformations responsively.

Rather than running all the computations specified in the registry proactively on all possible upstream data, a well-implemented and efficient feature store runs transformations on demand to compute data that has not already previously been computed and persisted in its stores.

When a feature store receives a query, it determines which feature values are already computed and stored. For the values that need to be freshly computed it determines the registry definitions that map to each feature value, runs those specified transformations, and then returns and persists the requested feature data.

Storing

Machine learning teams require two forms of data access: real-time, to serve modeled or computed results to customers (for example, a bank might want to know whether a transaction that just happened is fraudulent and an ecommerce company might want to recommend content to a user based on the products they've just looked at), and **retroactive**, to generate data for model training, to monitor how features are changing over time, or to run analytic queries.

These are incredibly different access patterns, but their outputs need to be consistent—the data used to train a model must look like the data used to make predictions. Combining them under one interface guarantees this consistency and improves efficiency. Feature stores like Chalk accomplish this through two abstractions: an **online store** and an **offline store**. The online store is a key

computed. It can also be queried (for instance, to access historical data).

Serving

Feature stores typically provide an interface for requesting data from either the offline or online store. At a high level, access to a feature store is divided into **online queries** and **offline queries**. The goal of online queries is to return the latest value for a feature as quickly as possible (either by returning the values from a cache or by running the necessary resolvers to calculate the requested features). The goal of offline queries is to either:

1. Retrieve historical data that has already been calculated,
2. Warm the online store cache (through an offline-online ETL process), or
3. Run batch jobs to generate features that don't need to be served through the online store.

There are multiple ways to run offline and online queries, but some of the most common methods include:

- Through **clients** (implemented in different programming languages),
- By making HTTP requests to a REST API,
- Through scheduled orchestration of queries, which will run queries on a schedule similar to scheduled ETL jobs.

Monitoring

Monitoring is a critical, yet less rigorously defined, part of feature stores. Knowing how your features and resolvers are behaving (or misbehaving) can surface previously invisible or inaccessible information, revealing subtle bugs early.

Because feature stores centralize feature computation, they can provide a comprehensive view into your features, including: the latency of your queries, the relationships between your features, the consistency of a feature's distribution over time, and the number of times a particular resolver is being run.

A good feature store allows you to define very granular metrics and connect them to your existing monitoring and alerting systems. For instance, if you have a feature defining whether a transaction is likely fraudulent (e.g. Transaction.is_fraud), you could configure a monitor that alerts an external service (like PagerDuty) if the percentage of queries marking transactions as fraudulent is suspiciously high or low. This allows instant visibility into critical decisions made by your ML system that otherwise might be hard to detect.

Additionally, some feature stores make it easy to export logs to common metric collectors, such as Prometheus, OpenMetrics (Datadog), and NewRelic.

Experimenting

A well-implemented feature store lets you experiment and collaborate on features and pipelines. Typically, clients create multiple environments for development and testing for their feature stores. This allows feature transformation code to be tested and evaluated before it is embedded in production pipelines. In the same vein of testing in isolated environments, feature stores can also enable engineering best practices for collaboration through isolated deployments within an environment, similar to version control, allowing for concurrent iteration.

Conclusion

Feature stores have become critical to machine learning platforms at companies of all stages, in all industries, and all sizes of ML and data teams. If your team is looking to ship production-grade machine learning confidently and quickly, a feature store is table stakes.

Chalk is data source agnostic with clients in multiple programming languages and lightning-fast deploys and queries for development. For a best-in-class