The Comprehensive Guide to Feature Stores

The art of feature engineering to take your machine learning projects to the next level

SEAN OWEN, MANI PARKHE, AND MARZI RASOOLI

Contents

Introduction 3

Feature Architecture 8

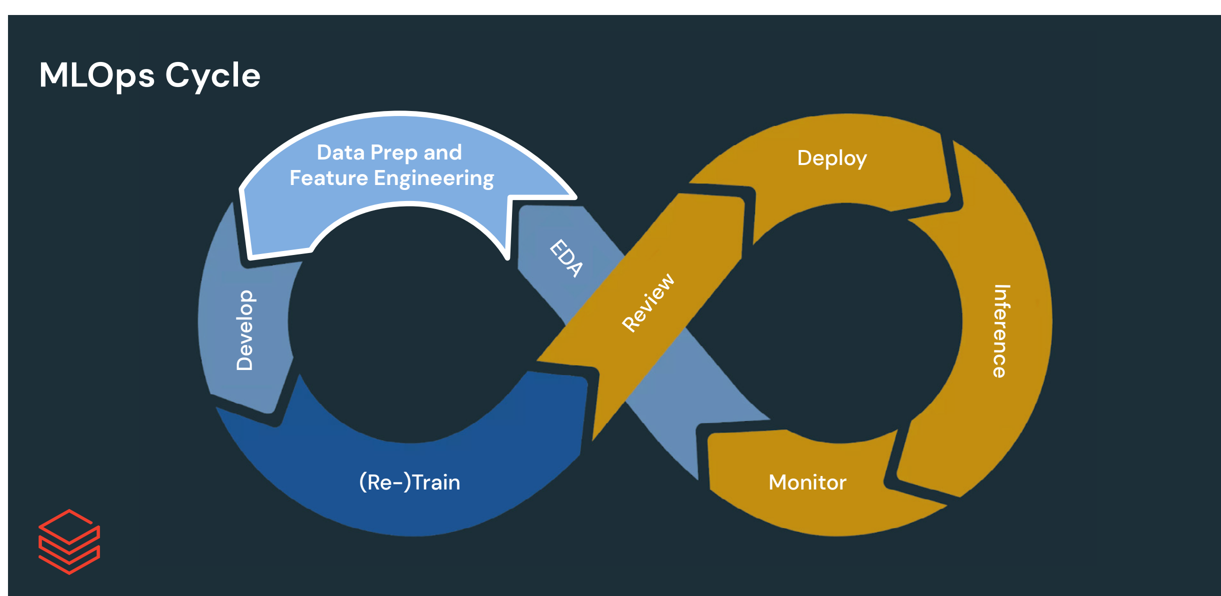
Operating a Feature Store 15

Organizational Issues 25

The Data Lakehouse: One platform for the full machine learning lifecycle 27

Introduction

Long before there was DevOps, there was just software development. Recently, a major trend in enterprise architecture has been MLOps, following the now-ubiquitous rise of machine learning, or ML. It's a sign that the practice of machine learning and data science has arrived in the mainstream enterprise when many start to ask, "How do we *operate* these models in production?"



Machine learning and data science depend on data. The rise of MLOps reopens long-standing questions of data management and operations, with new urgency. Thus the latest front in MLops is the *feature* store.

A feature store manages *features*, or input data to a machine learning model. At first glance, it doesn't seem different from the management of data in general. After all, databases with tables of data are nothing new. In practice, machine learning and data science share some needs (reliability, lineage, versioning) with applications that consume data and not so much others (transactions). The emphasis is different. Given the importance of machine learning, its particular needs (such as requiring data at both training and inference time) justify a new class of data management tooling.

The problem with features — and the purpose of feature stores

Machine learning models don't simply consume raw data. The functions of raw data are what's meaningful for a learning problem. In a model that predicts customer churn, for example, one might find:

- Aggregations of raw data over time windows, like trailing 7-day purchases
- Joined combinations of data sets, like customer demographic information joined to transaction features
- Complex functions of customer information, like estimated customer lifetime value

The process of creating these values from data is *feature engineering*.

DISCOVERY

Those features can be shared and reused in different models, and you can bet that they *will need to be* reused. Teams can't reuse what they can't find, so one purpose of feature stores is **discovery**, or surfacing features that have already been usefully refined from raw data.

LINEAGE

Sharing entails dependency, however. Reusing a feature computed for one purpose means that changes to its computation now affect many consumers. Feature producers need to understand the **downstream lineage** of features. What models and deployments of models depend on it? Likewise, feature consumers need to understand the **upstream lineage** of a feature to reliably use it. How is it computed, and who owns it?

SKEW

The problem of lineage isn't new or specific to machine learning — however, ML presents a different problem: managing feature transformation logic. Engineering features means executing logic to transform raw data, but this happens in two possibly quite different contexts: training the model and applying it to new data (inference, or scoring). Models may be trained in one environment, like Databricks, with access to scale-out distributed computing, SQL engines and secure connectors to offline data sources. They may be deployed, and called, from another type of environment entirely, like a Java web application using the model as a service.

Once you've built your model, reproducing the necessary input data and data transformations logic for inference may be nearly impossible, because model training and production tech stacks could be entirely different and managed by different teams. Where it's possible to reproduce the logic, it may be difficult to keep the two versions of the logic in sync, or the copied logic may yet not execute fast enough if the model is used for real-time sub-second-latency inference. If done incorrectly, the best-case scenario is a runtime error; the worst-case scenario is a silently wrong model continuing to make predictions.

This problem of **online/offline skew**, or the difference between the inference and training environment, is somewhat unique to machine learning, and appears quickly once teams move to production. Managing input data can have its own challenges. In a large organization, it can be difficult to guarantee that the source of data used for batch feature computation is the same upstream source used in inference at real time. When multiple teams manage feature computation and ML models in production, minor yet significant skew in upstream data at the input of a feature pipeline can be very hard to detect and fix.

NOTE: Databricks Feature Store by design solves this data and compute skew problem! Win!

Instead of aiming to make arbitrary featurization *logic* portable and fast, which would be almost impossible in the general case, feature stores typically aim to make the features portable — that is, the data itself. Hence, feature stores are data management tools at heart.

The need for a feature store depends on the level of automation required by your use case and organization. Anyone productionizing machine learning models will probably encounter the problems a feature store solves. So now you might be asking, how should one be designed? Are there new data architecture questions to answer? Versioning? Data access? Performance?

In the following sections, we highlight key questions that arise in designing and deploying a feature store, and try to offer some answers. They are written with the Databricks Feature Store in mind, but many ideas and principles apply to any similar tool.

What is a feature?

To architect a feature store and its contents, it's important to understand what a feature is in the feature store paradigm. It's reasonable to picture a feature store as just a database at heart, with more functionality on top and alongside (see below). Feature stores have feature tables of rows, with typed columns, storing feature values.

Although feature stores can accommodate unstructured data, they do adopt a tabular paradigm for data, as most machine learning models' direct input is structured (even if possibly derived from unstructured data).

For example, consider a model predicting customer churn. It relies on information about customers and benefits from receiving customer-related features as input.

customer_id	tenure	est_lifetime_value	7_day_calls	...
473337	2	901.32	13	...
480801	6	5828.80	7	...
...	...	...	...	...

Features are meant to be reused by combining them with other data sets. To accomplish that, there must be some means of identifying how to join the features, and so feature tables **need a primary key**. The values must be associated with something uniquely identifiable. In the customer example below left, customer_id would be the natural candidate for a primary key.

Isn't everything a feature?

Should all data related to data science and machine learning be managed in a feature store? No. A feature store adds value in managing transformed data **suitable for direct use in a machine learning model,** not the raw data that is being transformed. That customer churn prediction model may learn, ultimately, from raw customer transaction records. However, transactions are not features. But functions of the transactions may well be — for example, total calls over time. Features are **derived values**.

The feature store paradigm may be described as "feature engineer once, reuse many." Feature values are computed once, stored, managed and shared. Features are **computed ahead of time**, decoupling computation of features from their usage. Computing potentially expensive features once can save cost and time.

A feature store's role in an architecture

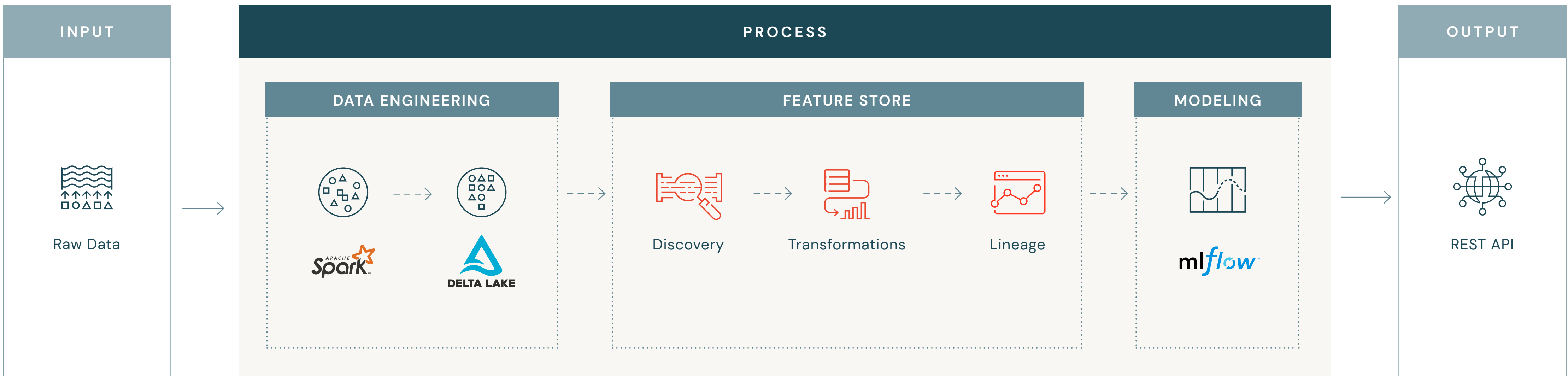
A feature store manages features, but its task depends on several related concerns:

- Its data needs to be stored somewhere
- The features are produced by transformations, which need to be executed somewhere
- The features depend on source data, which need to be accessed from a data store or stream
- Models require the feature values at both training time and inference time

Databricks and other standard open source tools can handle these.

- Feature data is stored in a Delta table under the hood (Delta is an open source project enabling the lakehouse architecture on top of cloud storage)
- Featurization is defined by code and scalable transformations with Apache Spark™ (i.e., not configuration-driven or using a custom DSL)
- Source data is read with Spark, meaning most anything can be read: other tables in a data warehouse, CSV files, XML, images, Apache Kafka streams and so on
- A modeling process based on MLflow (an open source platform for the MLOps lifecycle) can log models in a "feature aware" way that makes them capable of looking up features at runtime

As a result, a feature store isn't necessarily a strange new addition to an enterprise architecture. The Databricks Feature Store in particular fits naturally with workflows already leveraging MLflow and Delta.



In this simplified architecture, elements in red highlight the feature management roles covered by a feature store. From left to right:

- Data is read by Databricks, connecting to standard data formats, stores or streams
- Raw data in so-called Bronze tables is ETL'd with open source tools like Apache Spark
- This and the resulting Silver tables are managed in the open Delta format
- The Databricks Feature Store runs feature engineering transformations built on Spark — in batch or streaming — on Silver tables, and stores resulting feature tables in Delta
- The Databricks Feature Store feeds data and feature information to a modeling process
- Open source modeling libraries are used to build models, possibly on Spark, and are managed with open source MLflow
- The model is deployed, whether as a batch or streaming job using Spark, or as a REST service (inside Databricks or on Kubernetes, etc.)

In this view, the Databricks Feature Store simplifies what might otherwise be a standard feature engineering pipeline, feeding a standard model tracking and deployment process. Adopting it doesn't entail much change in either of those approaches, but instead purposefully integrates with them. The Databricks Feature Store eliminates the need to implement manual tracking and monitoring of feature creation and exploration, and enables out-of-the-box reuse.

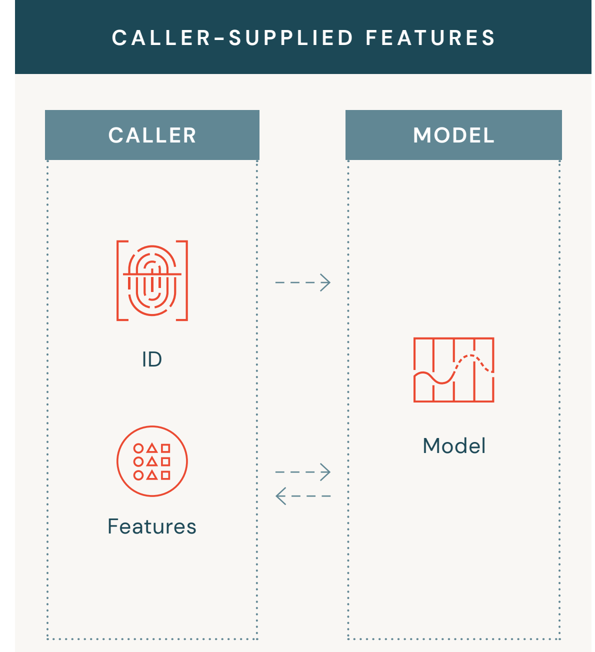
Feature Architecture

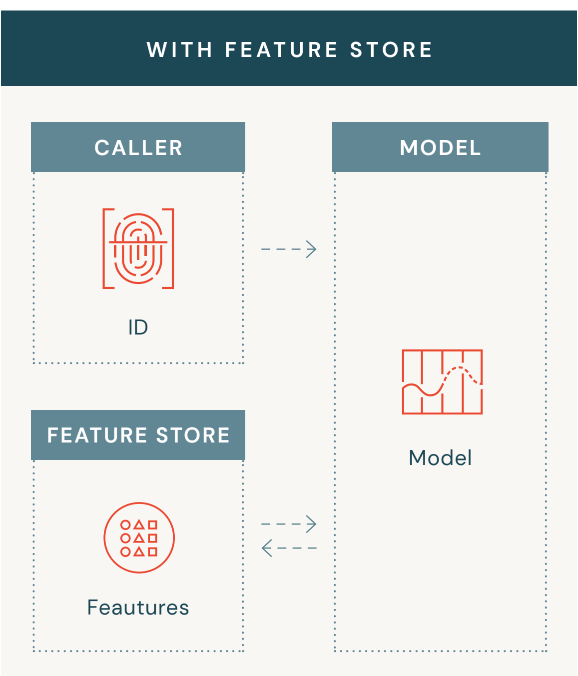
Runtime inputs vs precomputed inputs

If a feature store can manage precomputed values that feed a model, then a good feature store should be able to help models find and load those values automatically. The Databricks Feature Store does this and makes an explicit design choice to integrate directly with model tracking and deployment managed by MLflow. The features that a model needs become an implementation detail of the model, and not something exposed to the caller. The caller does not, for example, need to load features on the model's behalf and hand them to the model. The caller is not even aware of what features are required.

This affects the model's input schema, and for the better. Because models are often treated as services, produced in one context and consumed in another, it might be better to call this the model's "API," because that is how it behaves. The less the caller has to marshal to invoke this model API, the better.

That is, a customer churn model might need many features as input: transaction totals, demographic information, services used. However, as a service that predicts whether a customer will churn, it would ideally take just one input: a customer ID. Why should the caller have to supply more than this?





This helps decouple the model from its caller. If the model controls what features it needs to look up and add, then it may more easily vary its input features without requiring updates to all callers. Those callers would otherwise have to update at the *same time* that an updated model is deployed if the model needs new features.

It may also be advantageous to avoid making the caller create and supply features. Callers deployed "at the edge," outside of an enterprise architecture, might otherwise require direct access to raw data in order to compute features. Reading and sending that data might be slow, costly or insecure.

Why isn't everything a feature?

If *everything* the model needs as inputs were managed as features, then the input to the model could become just an identifier, with the model looking up what it needs by that identifier. In this customer churn model, the only thing needed at inference time would be a customer ID.

Runtime inputs

Unfortunately this isn't usually possible. The model's prediction may depend on facts that are only known at runtime, not ahead of time, and thus not precomputable as a feature. For example, in the context of a customer churn model, it may be useful to know whether someone who had called customer service recently escalated the issue to a manager. This can be known and recorded after the fact, and used to learn from later, but it is not a previously known fact that can be looked up from the feature store when deciding whether the customer on the phone right now with a manager is likely to churn.

This is an example of a model input that can't be managed as a feature for this model. The caller has to supply it, because it is not known ahead of time. It's possible that this piece of information becomes a feature in the context of another model, where the last-known value recorded ahead of time is just fine.

Why couldn't the support call escalation be recorded quickly as a feature and looked up? It could, but it's unlikely to be a useful idea. There is latency. Information can make its way through a streaming architecture to a job that computes, stores and syncs new feature values only so fast. Even though that could happen in seconds, that may not be fast enough in the context of a model serving real-time requests. Even if it is fast enough, it's probably undesirable for the architecture to go to the trouble of storing and rereading a fact that the caller already knows and can supply.

In short, choosing what to store and use as a feature has effects on the "API" of the resulting models, and in a good way. How models are used will determine what makes sense to store as a feature.

Labels as features

Should ground truth labels be managed in a feature store? These are not inputs to a model, but the correct outputs. In that sense, they do not seem to belong in a feature store, which is primarily concerned with managing model inputs.

They could be managed in a feature table. Some attribute of a customer, for example, that is to be predicted by one model could conceivably also be an input to another type of model. Perhaps customer lifetime value is predicted by one model, and that value is used as input to a churn model. The output from the first model could be written to a feature table and managed as input to another.

In a situation like this, it's important to not inadvertently use a label or features derived from the label in a model predicting that label.

Features for unstructured data

Feature stores adopt a tabular paradigm, where features are organized into tables with typed columns and even a primary key. How do so-called unstructured data fit into this seemingly structured design?

Unstructured data isn't really unstructured, it's just not tabular. This term typically (and misleadingly) refers to images and text. Of course, machine learning is known for working wonders on images and text, learning to classify or even generate novel pictures and articles. They need a place in a tool designed to support machine learning.

The problem isn't whether unstructured data can be stored; they at least can be encoded as bytes, and bytes could be stored in a feature store table, like any database table.

However, unstructured (or simply differently structured) data is "raw" data in the feature store paradigm, like the individual transactions that might feed that customer churn model. They're valuable data, but they don't offer reusable, derived information.

Note that models that consume text or images don't really handle them as bytes anyway. A key step in many deep learning models is to learn an "embedding" of such data. An embedding is a vector, or list of numbers, that usefully summarizes the input in some way. It can condense a large, complex text document or video input, for example, into a compact vector that is more meaningful for learning tasks and ready to feed into a model. However, an embedding can be expensive to compute.

Therefore, embeddings of unstructured data are good candidates for features. For example, a company that maintains the text of a user's forum posts might embed the posts and save those embeddings as features, as a useful summary of the posts. Many machine learning tasks that need to learn about forum posts could then reuse the embedding.

Architecturally, embeddings are just arrays of floating-point values and can be stored as such. There is nothing special about this type.

Handling time dimensions

Features are aggregations over raw data and describe the characteristics or behavior of an entity that the feature represents. Invariably these feature values vary over time, and as such, features are inherently time series data. It's easy to overlook this time dimension when reasoning about a feature store's design. Typically, for online model scoring, it's only the latest value of features that matters, and in order to ensure high accuracy of the model, the online feature stores are typically kept up to date, with the latest feature values accessible for model scoring. In some cases, the latest feature values are computed on the fly from raw data, which may be available in online data sources. However, doing so may lead to data skew, as previously discussed. For this reason, these latest feature values are published for the offline feature tables to online stores. That raises the question: Is it sufficient to store only the latest feature values in offline feature tables, or should it store all historical time series values for features?

When training a model, using only the latest feature values may lead to inaccuracies. For example, let's consider a data scientist training a customer churn model with the last two years of data from various data sets, such as user interactions (clicks, purchases, returns) and user communications. If the training pipeline joins this data with only the latest feature values for each customer, it would cause the training code to see incorrect data, since the latest features may already have included the effect of the user interactions in the past. This is referred to as "data leakage," and in order to avoid this, the data scientist needs to employ techniques to ensure that each training row is joined with that user's feature values at the event time for that user interaction. Hence, when designing feature tables to store time series features, it is important to record the 'event time' or 'timestamp' or express the time dimension in addition to the primary keys. Let's explore different techniques used to achieve this.

Native technique: Handling time series using a timestamp column as primary key

When using features from a previous point in time for model scoring or training, one of the techniques employed is to define a column that encodes the event time. Feature values would be computed and stored for many points in time, not just the current latest value. For this, a user can create a table with a timestamp or date column as part of the table's primary key. In the customer churn example, (customer_ID, 'date') could be used as the primary key, and feature store logic could produce features per customer per month. At inference time, a date would have to be passed along with the customer ID, at least, in order for the feature store to know which feature values to retrieve.

customer_id	date	tenure	est_lifetime_value	...
date	2021-11-01	2	901.32	...
tenure	2021-11-01	6	5828.80	...
est_lifetime_ value	2021-12-01	7	5703.31	...
...	...	...	...	...

However, it would be inconvenient if time-based keys were treated like other primary keys and the feature store lookup required an exact match on the time value. A caller would have to know exactly the time at which the desired features were computed when presumably the caller just wants the latest value as of that time. This doesn't match the simpler case without time-based keys, where implicitly the latest value is retrieved, no matter when it was computed.

Time series feature tables

For that reason, the Databricks Feature Store supports time series feature tables. When creating one, you can specify an optional column that represents the time scale. This is a first-class field of the feature table, called a timestamp key. Internally, this column will be used along with the primary keys to determine data uniqueness and for "merge" semantics when adding new features to the table. However, in the feature lookup workflow, these timestamp keys will be used in native "as of" joins. Given a time value, the join will match the latest time in the table that isn't after the given time value.

So, in the churn example, a caller may pass a customer ID and a time, and the model would look up the latest feature values for that customer "as of" the given time.

NOTE: In the churn example, the timestamp key (date) is of DateType. The Databricks Feature Store supports specifying a timestamp key of either DateType or TimestampType (for specifying more granular event time). In the latter case, an "as of" join will work in the same manner to look up the latest feature values for that customer "as of" the given time.

Using a particular time in the past for inference is an unusual use case, but it's useful for backtesting a model with feature values as of a particular time in history. However, time is still important as a dimension in a feature table even if you only query for 'now'.

Historical feature values

It may be important to retrieve previously computed feature values or get the state of a feature table at the time of model training or inference for auditability or reproducibility purposes. Thankfully, this does not require any extra consideration in the Databricks Feature Store. Because it is built on Delta tables, which keep track of changes to the underlying data via the transaction log.

CAUTION: Looking up historical feature values will use the compute timestamp for the feature table and provide a snapshot of the table as it was at a particular time in the past. While this can be used for the purpose of auditing or querying a snapshot at a single point in time, this mechanism has several limitations when used to train models with features as of a particular event time in the past.

Features over time during training

Putting that aside, time may still be important in a feature table to support model training. Consider that in a customer churn model, much more is known about a customer than the current state. There is a whole history of states. At many points in time, the customer did not churn — until the customer (perhaps) did. When using this data to train a churn model, it is important to use

all of that information so the model can train from examples that caused the users to not churn and differentiate from examples that caused churn. For this purpose, a data scientist may choose to use data sets such as user interactions (clicks, purchases, returns), user communications and other activity-type data sets as the main sample points for training and to join user features as of the event time in each of the data sets.

The timestamp key is not returned or used in training, and typically it would not be. However, it becomes necessary as input at inference time nevertheless. Callers will supply the time as of which the inference should be made, and this could simply be the current time/date.

Time dimensions and online stores

Time series feature tables in the batch layer (Delta) can grow large, with a row per customer and month — not just per customer — in this example. The historical data is, of course, important for model training. This can also be required for batch scoring where necessarily older data is being scored and the model expects the feature values "as of" the time of that batch of data. However, if inference is always made as of "now" (typically in online scoring), then historical feature values are unnecessary at inference time.

While an offline store based on Delta can scale easily, this may present a challenge for the online store. When publishing features from time series feature tables to online stores, the Databricks Feature Store will only publish the latest feature values (using the timestamp key) for each primary key combination. This new batch of "latest" feature values is then merged into the online store such that there is only one row per unique primary key value. In the future, we will support multiple historical snapshots in online stores.

Grouping features into tables

Of course, a feature table of customer-related features is likely keyed by a customer ID. A feature table of store-level sales data might be keyed by a combination of region ID and regional store ID. These two must be separate feature tables, of course, as they contain information about different entities. The key often determines what features could, or could not, exist in a single feature table.

Could two different feature tables contain customer-related features, keyed by customer ID? It's possible. Which to choose is a data architecture decision, just as with any database tables. A few factors influence that decision.

Security

Some features may be considered sensitive information, such as a customer's income or age. Other information might require less control, like a count of customer transactions. Some use cases and teams may not have access to more sensitive information for policy or regulatory reasons.

Enforcing different access controls for different customer information is simplest when the sensitive data exists in a different feature table. Table-level access is easier to manage, and at this time, the Databricks Feature Store does not yet support column-level ACLs. For example, maybe customer income and age are stored in a tightly controlled table, while average transaction size could exist in a less-controlled table.

Source

One can imagine customer features originating from many sources as well. Transaction logs may produce transaction-related features like total spend; web logs might provide activity-related features. The data pipelines that consume these sources might be separate jobs, even managed by separate teams.

It's more natural for separate pipelines to feed separate feature tables. Data engineers manage a transaction data pipeline that populates and updates a transaction feature table, rather than a job that tries to update one large central table that many teams are updating.

Performance

Separating features into tables can be more performant at scale too. Imagine that some customer data is slow-changing, like age or income. Some data is fast-changing, like daily average usage. Segregating the fast-changing features into a separate table could be more performant to update frequently, rather than updating one table that also contains features that rarely change. This is more relevant when considering the online store, to which features are synced. The Databricks Feature Store syncs feature tables to an online store at the table level. If all customer features are in one feature table, the whole table is replicated online, even if only subsets of it are updated.

Operating a Feature Store

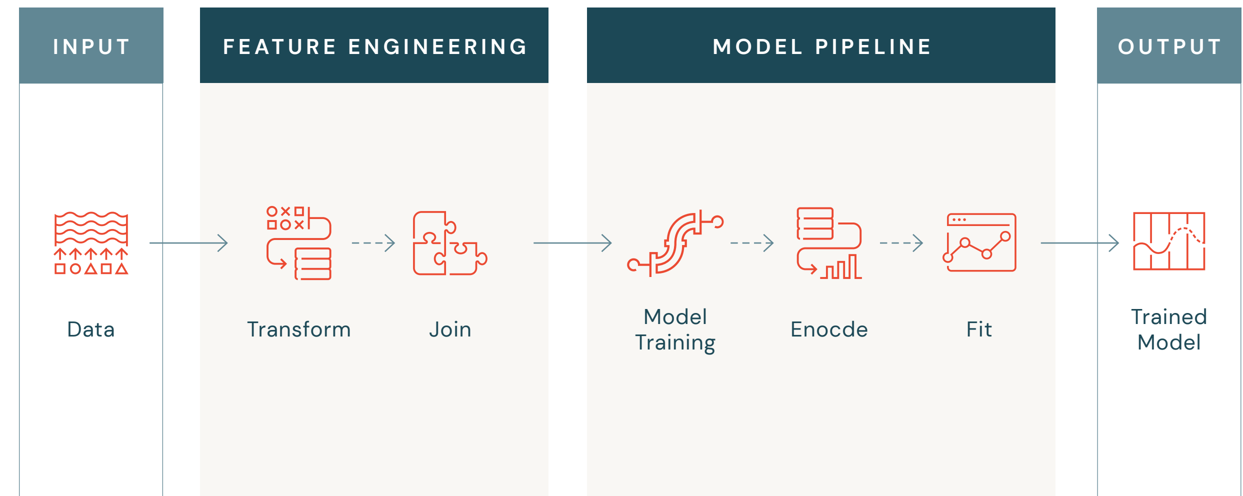
Migrating from existing feature storage

The audience for a feature store is typically a team or organization that has already begun to develop and deploy models, and so likely has some strategy in place for managing features, even if only an ad hoc one. To build a model at all, feature engineering has to happen somehow.

A basic training pipeline almost always has a few essential components:

- Loading and transforming data with tools such as pandas and Spark
- Joining transformed data with other data sources
- Fitting an ML pipeline, which:
  - Applies model-specific transforms (e.g., pretrained embedding, one-hot encoder)
  - Fits a model

That is, the data transformation is often split into a portion that happens outside of model training, and a portion that can happen within the model's own "pipeline" abstraction. The most common examples are scikit-learn, whose Pipeline abstraction and transformers are frequently used with scikit-learn and other libraries, and Spark ML's similar Pipeline abstraction. Deep learning frameworks like Keras or PyTorch likewise allow some preprocessing in a model with the addition of layers that normalize, resize, etc.



It would be convenient if all featurization and encoding could live entirely within a modeling tool's pipeline. Then the featurization logic would all travel with the model artifact. However, most pipeline libraries only support simple transformations like scaling, imputation, etc., and it's difficult to extend them to support custom transformers. While possible to implement custom transformers, it is extra work, and typically rules out using tools like Spark for large-scale transformations (e.g., Spark can't be used in a scikit-learn pipeline). It again raises issues of managing that code's dependencies and performance at runtime.

If you're moving to a feature store like the Databricks Feature Store, how does a training workflow like this translate? Generally speaking, it's the feature engineering that comes before model (pipeline) fitting that migrates to a feature store. The portions inside a pipeline abstraction can stay.

Why? The transformations found in a pipeline are also likely to be specific to the model itself, like applying scaling that was fit to the model's training data. That is, it may not make sense to manage a set of scaled features created for one model, as it isn't usefully reusable in other models.

Migrating the loading and transformation data to the Databricks Feature Store can be simple, even very simple, depending on the tools already in use. For example, if the transformations are already expressed as Spark code, then it's only a matter of wrapping up that code in a function and invoking it slightly differently, because the Databricks Feature Store just relies on transformations expressed in terms of Spark:

Before

```
raw_data = spark.read...
features = raw_data.
select(...).agg(...)...
...
features.write(...)
```

After

```
fs = FeatureStoreClient()

def compute_features(raw_data):
  features = raw_data.select(...).agg(...)...
  ...
  return features

fs.write_table(..., compute_features(...))
```

Alternatively, the transformation code might be expressed using pandas, another popular Python package for data manipulation. This code has to be ported to Spark to work with the Feature Store API. This can be relatively simple now that Spark 3.2.0 supports the pandas API on Spark (formerly known as Koalas).

It's possible, even likely, that an existing machine learning pipeline already writes features to some table. If so, those existing tables can be registered as feature tables instead of creating a new one (see register_table instead of create_table). Note that in this case, it's not advisable to continue writing to the table directly without using the Feature Store APIs, as the Databricks Feature Store is trying to manage and track writes to and reads from the table, as well as guarantee things like key uniqueness.

Any logic to join results from other data sources then becomes FeatureLookups in the Databricks Feature Store, instead of manual joins:

Before

```
input_df = spark.read...
training_set_df = input_
df.join(table, on=key)...
join(...)
```

After

```
fs = FeatureStoreClient()

# input_df only needs to contain the key and target
input_df = spark.read…
training_set = fs.create_training_set(input_df,
    [FeatureLookup(table_name = table, lookup_key =
key), ...],
    label=”...”)
training_set_df = training_set.load_df()
```

Finally, migrating the modeling process to take advantage of loading features from the Databricks Feature Store is, hopefully, mostly an exercise in refactoring into simpler code. Modeling proceeds as usual, including creating and fitting pipelines; this does not change.

To track the resulting model in a Feature Store-aware way, it's necessary to slightly change how MLflow is used to log the model. (And if you're not using MLflow, you should! It's necessary in order for the model to transparently join features it needs at runtime.) The MLflow model logging method is replaced by a wrapper in the Databricks Feature Store client instead:

Before

```
with mlflow.start_run():
  ...
  model = pipeline.fit(..., …)
  mlflow.log_metric(...)
  ...
  mlflow.sklearn.log_model(model, “model”)
```

After

```
with mlflow.start_run():
  ...
  model = pipeline.fit(..., ...)
  mlflow.log_metric(...)
  ...
  fs.log_model(model, “model”,
    flavor=mlflow.sklearn, training_
set=training_set)
```

At inference time, any code that recreates or reinvokes feature transformation logic to prepare data for the model to score simply goes away. The model, logged by MLflow and wrapped up with logic to look up necessary features, can be applied conveniently with the Feature Store client:

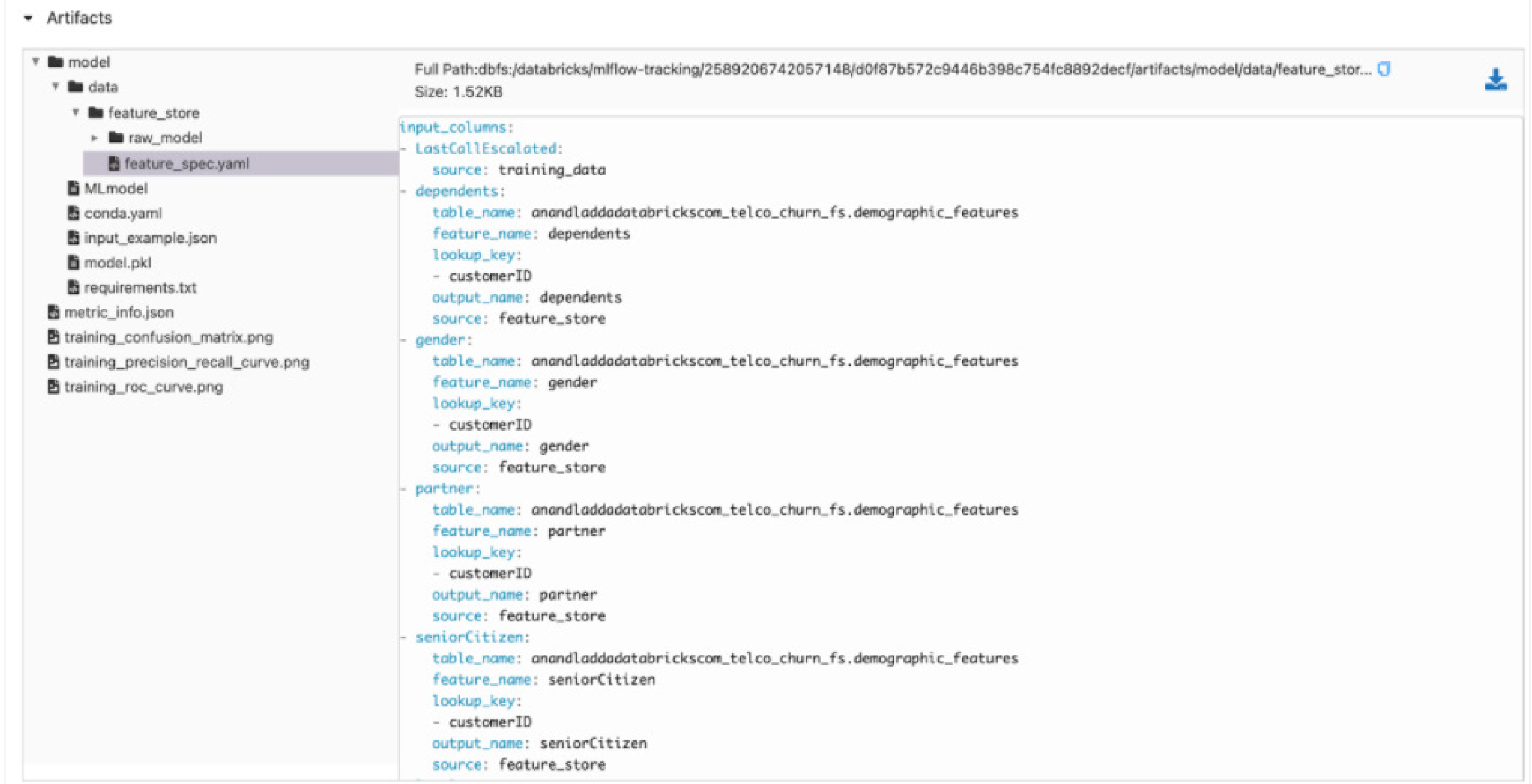
Before

```
model_udf = mlflow.pyfunc.spark_udf(spark,
“models:/my_model/production”)
raw_data = spark.read...

# featurize again
transformed = raw_data.
select(...).agg(...)...

# join again
joined = transformed.join(table, on=key).
join(...)
predictions = joined.
withColumn(“prediction”, model_udf(*cols))
```

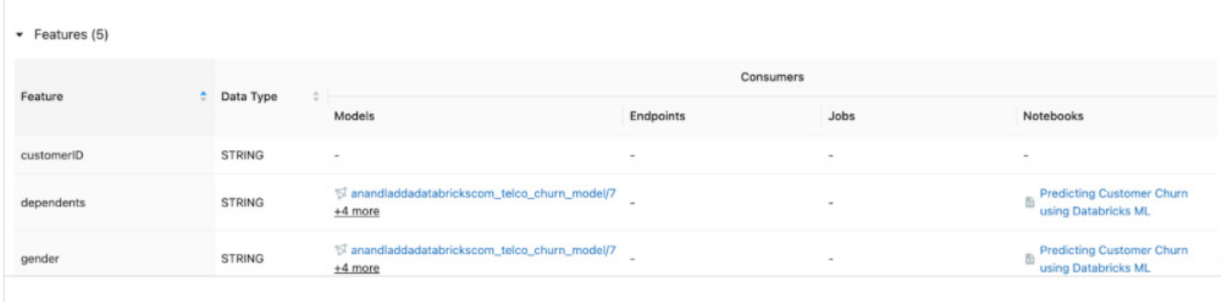
After

```
raw_data = spark.read...
predictions = fs.score_batch(raw_data,
“models:/my_model/production”)
```

This is simpler, and the magic lies in how models are logged with the Feature Store client. Take the following example of a model logged with fs.log_model():



In addition to the serialized model, a feature_spec.yaml file describes the feature lookups required to reconstruct a row at inference time. The model now possesses information on the features it needs, and conversely, the features possess information on the models they serve. If we look at the Feature Store UI, we see which feature columns are associated with models that have been promoted to the MLflow Model Registry:



Features (5)

Feature	Data Type	Consumers			
		Models	Endpoints	Jobs	Notebooks
customerID	STRING	-	-	-	-
dependents	STRING	anandladdadatabrickscom_telco_churn_model/7 +4 more	-	-	Predicting Customer Churn using Databricks ML
gender	STRING	anandladdadatabrickscom_telco_churn_model/7 +4 more	-	-	Predicting Customer Churn using Databricks ML

The workflow described below left works for streaming use cases as well. You can write feature values to a feature table from a streaming source.

```
streamingData = (spark.readStream...) fs.create_feature_table(table_name, schema, keys)
# stream to feature table
fs.write_table(table_name, df=streamingData, mode="merge")
```

Or in the event that your inference data comes from a Spark structured streaming pipeline, the same feature store API can be used for inference.

```
streamingData = (spark.readStream...)
stream_predictions = fs.score_batch(model_uri, inference_stream)
```

You can also stream feature tables from the offline store to an online store with:

```
fs.publish_table(name=feature_table_name, online_store=online_store, streaming=True)
```

Promotion from dev to prod

Productionizing anything leads eventually to the age-old question: dev vs prod. How do artifacts move from development to production?

We take for granted that code is developed, tested and only then rolled out to production. Models need this, too, though it's more ambiguous: Are models pushed from dev to production, or is the code that builds the model pushed to production? And data is typically not "promoted" from dev to prod at all, in any sense.

A feature store like the Databricks Feature Store falls ambiguously into this spectrum, because it touches each of these elements. A feature store runs featurization code to produce data that is used to produce models.

Promote features as code, not data

One way or the other, featurization code needs to be tested, and so the development environment will use an instance of a feature store to test its logic.

Code that computes features is code, and can be tested, versioned and deployed like other code, from development to production. For example, featurization notebooks may be developed against a branch of a git repository, and when changes are tested and vetted, merged to a production branch in git and pulled into production.

That is, production runs featurization code on production data to produce production features. Production does not receive "promoted" features from development in any sense. The development environment will have a feature store for functional testing, but its data may not be used, or it might be used only as part of testing in development.

Promote features after data eng, before modeling

Features have upstream dependencies, such as data engineering outputs that featurization logic depends upon. Features, in turn, are depended upon by models used in inference. As with any other architectural component with dependencies, deployment to prod has to roll out in order.

Featurization code that depends on new data engineering outputs can't deploy before the new outputs are operating in production. And models that depend on new features can't be promoted to production before the featurization logic that produces them is rolled out.

The Databricks Feature Store tracks metadata about upstream and downstream dependencies to help developers understand these dependencies and make it easier for them to plan promotion accordingly.

Migrating feature definitions

In production, models are retrained as new data arrives with new information about the world. Data science doesn't sit still either, and teams also improve the accuracy of a model over time by altering the model itself or the data that it is fed for training.

Tools like MLflow's Model Registry manage successive versions of a logical model and provide workflow for the production promotion process. What about feature definitions? Teams may add new predictive features, or modify or fix existing definitions. This requires care, like any process change.

Adding features

For instance, in the customer churn example, a team may find that knowing the average price increase (or decrease) over the customer's tenure helps more accurately predict the customer's propensity to churn. This is the simplest case. The feature table definition would return the new column:

```
def compute_features(raw_data):
  ...
  features = feature.withColumn("avg_price_increase", ...)
  ...
  return features
```

To update existing customers with the newly computed value, use this merge mode:

```
fs.write_table(..., ..., mode="merge")
```

The underlying feature table will get a new column, 'avg_price_increase', on the next execution. Any customers passed as input to 'compute_and_write' will have this new feature value computed, and their rows in the feature table will be updated with the new value (just like other features). It's 'merge' mode that chooses to update, rather than append, new rows.

Like adding a new column to a database table, this "sometimes" doesn't cause problems for other consumers of the table, as it simply doesn't appear when not selected. However, just as a new column might disrupt callers depending on the result of 'SELECT *', it's important to consider whether other model pipelines are selecting all features from the feature table.

For example, a feature lookup on this feature table that selects all features will see this:

```
FeatureLookup(table_name="customer_features",
              lookup_key="customer_id")
```

Any model pipeline doing so will start to look up this new feature, use it in training and expect it at runtime. This is probably not desirable. Instead, consider having pipelines enumerate exactly what they look up:

```
FeatureLookup(table_name="customer_features",
                lookup_key="customer_id",
              feature_names=["...", "..."])
```

Deleting features

Consider if, later, the team finds that 'avg_price_increase' wasn't so predictive after all. Model pipelines would stop including it in training, and so would not look for it at inference time. Once all models stop using a feature, it would be safe to stop computing it. The compute_features function's definition can simply not compute and instead returns this as a column.

At the time of this writing, the API does not allow completely deleting a feature from a feature table. It could be overwritten with nulls, or ignored. It may, in fact, be desirable to keep it for later reproducibility of older models.

Of course, it's important to know if all models have stopped using the feature before deleting it. The Databricks Feature Store records which models are known to use a feature by recording which models are logged to MLflow via the Databricks Feature Store client's log_model method. If used consistently, this helps teams understand when a feature is truly unused. Note that the Databricks Feature Store can't track usages of features that do not proceed through its client.

Modifying features

Changing a feature's definition is more involved. Instead, maybe the team finds that 'avg_price_increase' was computed incorrectly. Fixing the logic is easy — just correct the code that computes the feature, and its next execution (with merge mode) will update the feature table with values computed according to the newest logic.

Existing models trained on the older definition have learned about average price increase based on a flawed computation, but fixing this input without retraining the model could actually make its predictions worse, as the model may have, to some extent, learned to correct for the error.

Synchronizing the release of the new computation with the release of a new, retrained version of every model that depends on the feature could be quite difficult. Ideally, models retrained on the corrected definition would use the new value, and existing models would continue to use the old value in the meantime.

This means that, for a time, the feature engineering pipeline must produce both old and new values. In the Databricks Feature Store, there is only one value of a feature at a given time. At the moment, there is no notion of feature versions. It's possible to achieve this effect manually, though a little inelegantly. Use a naming convention to separate new from old versions, and produce both. Here that means continuing to compute 'avg_price_increase' but also producing, say, 'avg_price_increase_v2':

```
def compute_features(raw_data):
  ...
  features = feature.withColumn("avg_price_increase",\
      ... old logic ...)
  features = feature.withColumn("avg_price_increase_v2",\
      ... new logic ...)
  ...
  return features
```

Of course, eventually the old feature may be "deleted" per above.

Incremental updates and backfilling features

So far, while it may be clear how to define featurization logic, and have the feature store execute it, persist results and record metadata about featurization, it's not necessarily clear where its input comes from. What is raw_data in the examples above?

This is caller-supplied in the Databricks Feature Store paradigm. The job that is invoking write_table decides what data to read, as this is an essential part of the overall featurization logic.

The input to the featurization logic has to contain all the columns that the code expects. What rows should be passed, though? It depends.

Case: New rows only

Perhaps the simplest case is one where the upstream raw data is not modified, only appended to. Featurization only has to happen once. It isn't necessary to reread past raw data, re-featurize it and rewrite the values to the feature store if the past data and the featurization logic haven't changed.

In this case, where the upstream raw data has a time dimension, it may be a matter of selecting the data with timestamp after the last featurization job ran. There are ways to figure this out in a Delta table as well without an explicit timestamp. This is also how the featurization job would work if the upstream raw data is a stream, as by nature only "new" records are arriving.

Featurization logic is often not that simple, however. For example, upstream data may be raw customer transactions. New transactions arrive for existing customers. This may mean that the latest value for customer features like "average spend" changes for those existing customers. The same idea applies, but the featurization logic may be updating, rather than adding new feature table entries.

Case: All rows

At an extreme, it's possible that some features are a function of all historical data. Imagine calculating customer lifetime value, which may be a complex function of all customers. If any customer data changes, all estimated lifetime values may change. In this case, it would be necessary to supply all raw data every time write_table is invoked.

It's always possible to rerun featurization on all data, recompute all features and rewrite updated values. This may be slow, but it won't be incorrect.

Many real-world use cases fall in points in between. Some features may be computable and updatable only based on current, new data, while some may require all data, and some may require a mixture.

It may be necessary to split featurization workloads into separate jobs, with some that compute based only on new data, where possible, and others that compute separately on all data each time, where not possible.

The problem of backfill

Consider a simple example of a feature in a customer churn model, like 'maximum call length'. As new raw call data arrives, it can only affect the maximum call length of customers that made a call. Typically these features can be computed only over newly arriving data as a maximum of the current max, and the length of the current call, for each customer.

However, consider the day this feature is added to the featurization logic. It will compute correctly for all customers that made a call, but will not have been computed for anyone else. This may be a problem; the feature needs to be "backfilled."

One solution is to run the featurization logic one time on all data to backfill. This is probably the simplest approach, as the featurization logic is just a function that can be applied to a different set of data in code.

This issue also arises when a feature definition is modified (see "Modifying Features" above). An updated feature definition might only be applied to new data going forward, and may require manually executing it on past data to update for all features.

Organizational Issues

Data science vs data engineering ownership

Feature stores address the problem of feature engineering, but feature engineering is adjacent to both the more general data engineering that precedes it upstream and the machine learning work that consumes features downstream. The role of the data engineer is fairly distinct from that of the data scientist. So, which group owns the feature engineering in between them?

Data engineering?

Features are generated from raw data that resides in files or tables. Data engineers are responsible for maintaining jobs to generate curated data sets. This includes developing, scaling and troubleshooting these pipelines.

In the customer churn example, data engineers might be responsible for making data available from different sources such as a mobile app, website or call center. Data engineers can ensure data is accessible, clean and reliable.

This partly also describes feature engineering — transforming data, possibly at scale, and troubleshooting transformation pipelines. There's a natural argument that it's just an extension of data engineering, as it will involve many of the same skills.

If so, lineage from a feature store framework would be crucial for data engineers to monitor in order to know how the feature data is used downstream.

Data science?

Of course, data engineers may know more about how to run feature engineering pipelines, but it's the data scientists that know what needs to be produced. Often, new features might be tried rapidly, so it could make sense for data science to own defining features and avoid round trips through another team to modify features.

Yet there is a difference between concocting a feature for an experiment and productionizing and sharing it. After all, the premise of a feature store is to publish useful maintained features. It's possible that data scientists iterate outside the feature store framework to evaluate features and work later with data engineering to move that code and featurization into the feature store. In the customer churn example, that is, if a data science team wants to experiment with 'log of average price increase' as a feature, it's not as if they need to alter the feature store to merely try out that value as a model input. They might temporarily generate and test this feature without using the feature store.

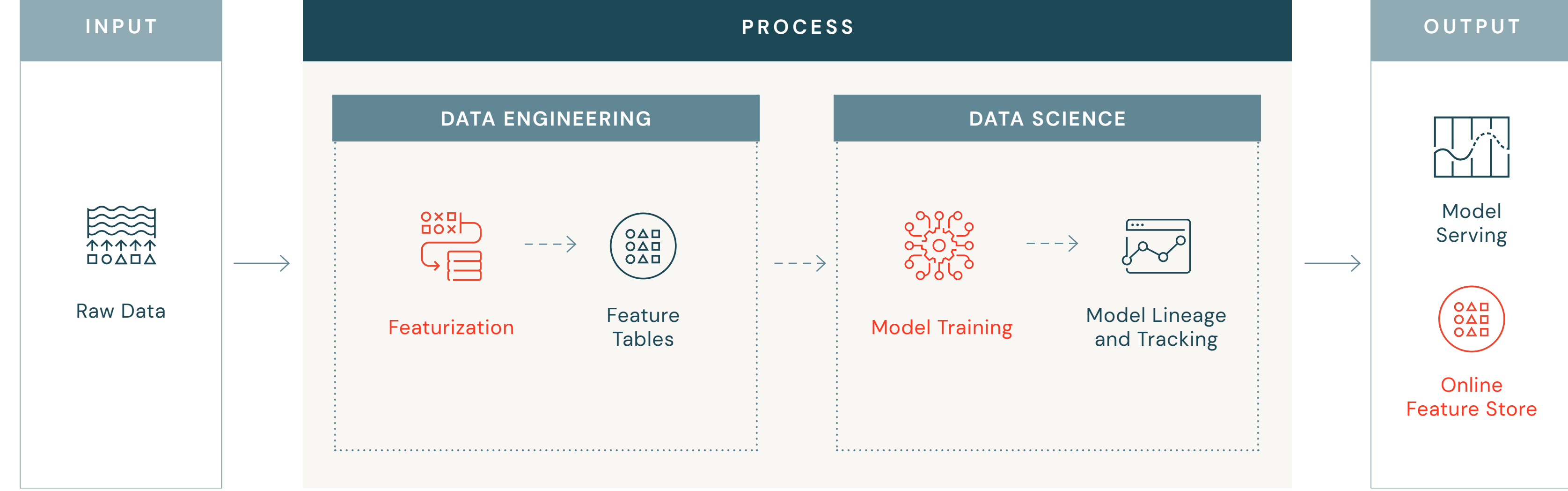
Data scientists may only want to define and prototype features, and hand them off to scale and harden, just as with data engineering in general.

Finally, to the extent a feature is shared, many data science teams depend on it, but there is only one feature store and pipeline generating it. It may be more natural for a group outside of any single data science team to own the features that several teams share.

The answer?

It depends, but on balance, the team that has the most familiarity with data pipelines should own feature engineering and the feature store. In large companies, that's the data engineering team. The feature store is a data store, and its contents often transcend individual data science projects. Data science teams can still iterate and experiment directly with data without involving data engineering every time, even if so.

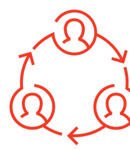
At right, the proposal is that feature store elements in red, like featurization and storage of features, be owned by data engineers. The feature store is still touched directly by data science pipelines downstream, to the right, and their downstream transformation logic may yet feed back into featurization pipelines owned by data engineering.



The Data Lakehouse: One platform for the full machine learning lifecycle — from data prep to production ML

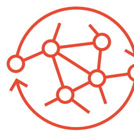
This leads us to an even larger discussion — which is outside the scope of this book — and that is, how do data engineers, data scientists and production machine learning engineers operate together, as a team, with the same data, within the same governance and security framework?

The answer is the data lakehouse. The Databricks Lakehouse is simple, open and collaborative, and trusted by thousands of the world's best companies, from large, successful enterprises to new-age digital-native firms. The Databricks Lakehouse is a unified platform that supports the full machine learning lifecycle — from data ingest, to feature engineering, to model training and tuning, all the way to serving and monitoring. It creates a space for data engineers, data scientists, and machine learning engineers to collaborate and build better AI solutions.



COLLABORATIVE EXPLORATORY DATA ANALYSIS

Bring data teams together and ensure your data is ready for machine learning

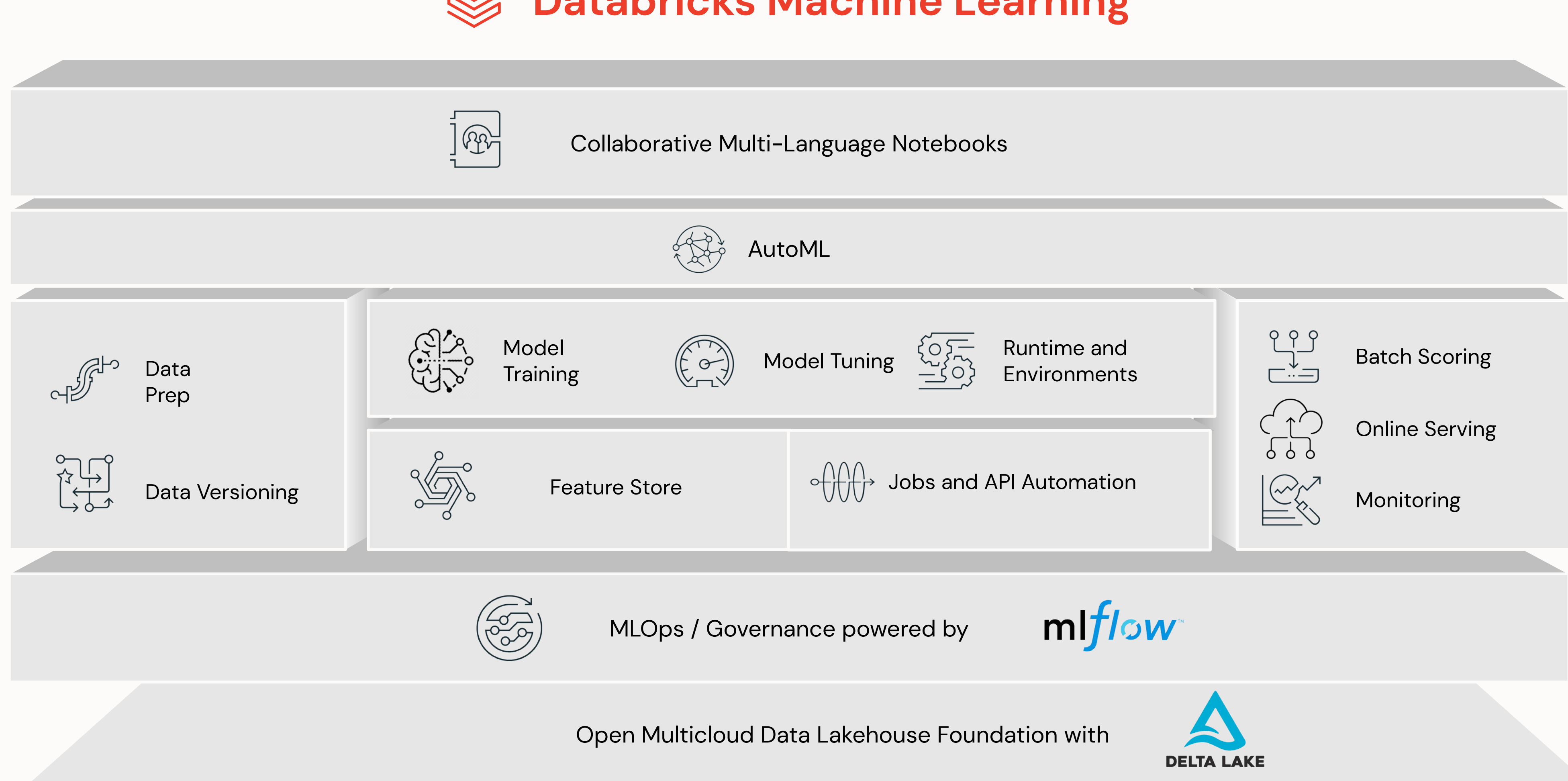


COMPREHENSIVE MODEL LIFECYCLE MANAGEMENT

Leverage a single platform across the full ML lifecycle for tracking, governance and reproducibility

RAPID, SIMPLIFIED MACHINE LEARNING FOR EVERYONE

Built-in no-code capabilities and AutoML help proliferate ML across the organization to a variety of personas



Discover how Databricks can elevate your data science and machine learning projects.

About Databricks

Databricks is the data and AI company. More than 7,000 organizations worldwide — including Comcast, Condé Nast, H&M and over 40% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark,™ Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

START YOUR FREE TRIAL

Contact us for a personalized demo
databricks.com/contact

© Databricks 2022. All rights reserved. Apache, Apache Spark, Spark and the Spark logo are trademarks of the Apache Software Foundation. Privacy Policy | Terms of Use