Chapter 2: What Problems Do Feature Stores Solve?

In the last chapter, we discussed the different stages in the **machine learning** (**ML**) life cycle, the difficult and time-consuming stages of ML, and how far we are from an ideal world. In this chapter, we'll explore one area of ML, which is ML feature management. ML feature management is the process of creating features, storing them in persistent storage, and serving them at scale for model training and inference. It is one of the most important stages of ML, although it is often overlooked. In data science/engineering teams in the early stages of ML, the absence of feature management has been a major hindrance to getting their ML models to production.

As a data scientist/ML engineer, you may have found innovative ways to store and retrieve features for your ML model. But mostly, the solutions we build are not reusable, and every solution has limitations. For example, some of us might be using S3 buckets to store features, whereas other data scientists in the team might be using transactional databases. One may be more comfortable using CSV files and the other might prefer using Avro or Parquet files. Due to personal preference and a lack of standardization, each model will probably have a different way of managing features. Good feature management, on the other hand, should do the following:

- Make features discoverable
- Lead to easy reproducibility of models
- Accelerate model development and productionization
- Fuel reuse of features within and across teams
- Make feature monitoring easy

The aim of this chapter is to explain how data scientists and engineers strive to achieve better feature management and yet fall short of expectations. We will review different approaches adopted by teams to bring features into production, common problems with these approaches, and how we can do better with a feature store. By the end of this chapter, you will understand how a feature store meets the objectives mentioned previously and provides standardization across teams.

In this chapter, we will cover the following topics:

- Importance of features in production
- Ways to bring features to production
- Common problems with the approaches used for bringing features to production
- Feature store to the rescue
- Philosophy behind feature stores

Importance of features in production

Before discussing how to bring features to production, let's understand why features are needed in production. Let's go through an example.

We often use taxi and food delivery services. One of the good things about these services is that they tell us how long it will take for our taxi or food to arrive. Also, most of the time, it is approximately correct. How does it predict this accurately? It uses ML, of course. The ML model predicts how long it will take for the taxi or food to arrive. For a model like that to be successful, not only does it need a good feature engineering and ML algorithm, but also the most recent features. Though we don't know the exact feature set that the model uses, let's look at a couple of features that change dynamically and are very important.

With food delivery services, the major components that affect the delivery time are restaurants, drivers, traffic, and customers. The model probably uses a set of slow-changing features that are updated regularly, maybe daily or weekly, and a set of dynamic features that change every few minutes. The slow-changing features might include the average number of orders a restaurant receives at different times of the day from the app and in person, the average time it takes for an order to be ready, and so on. It might seem like these features are not slow-changing, but if you think about it, the average number of orders might differ based on restaurant location, seasonality, time of the day, day of the week, and more. Dynamic features include how long the last five orders took, the number of cancelations in the past 30 minutes, and the current number of orders for the restaurant. Similarly, driver features might include average order delivery time with respect to distance, how often the driver cancels orders, and whether the driver is picking up multiple orders. Apart from these features, there will be traffic features, which change much more dynamically.

With many dynamic features in play, even if one of them is an hour old, the model's predictions will go off the charts. For example, if there is a crash on the delivery route and traffic features don't capture it and use it for inference, the model will predict that food will arrive more quickly than it actually will. Similarly, if the model cannot get the current number of orders at the restaurant, it will use the old value and predict a value that may be far from the truth. Hence the more up-to-date features a model gets, the better the predictions will be. Also, another thing to keep in mind is that the app will not give the features; the app can only give information such as the restaurant ID and the customer ID. The model will have to fetch the features and facts from a different location, ideally a feature store. Wherever the features are being fetched from, the infrastructure serving it must scale in and scale out based on traffic to efficiently use resources and also to accommodate requests at low latency with a very low percentage of errors, if any.

Just like the food delivery service, the model we built in the first chapter needs the features during inference and the more up to date the features are, the better the customer's **lifetime value** (**LTV**) prediction will be. Good predictions will lead to better actions, resulting in excellent customer experience, hence better customer affinity, and better business.

Ways to bring features to production

Now that we understand the need for features in production, let's look at some traditional ways of bringing features to production. Let's consider two types of pipelines: batch model pipelines and online/transactional model pipelines:

- **Batch models**: These are models that are run on a schedule, such as hourly, daily, weekly, and so on. Two of the common batch models are forecasting and customer segmentation. Batch inference is easier and less complex than its counterpart since it doesn't have any latency requirements; inference can run for minutes or hours. Batch models can use distributed computational frameworks such as Spark. Also, they can be run with simple infrastructure. Most ML models start as batch models and, over time, depending on the available infrastructure and requirements, they go on to become online/transactional models.

Though batch models' infrastructure is simple to build and manage, these models have drawbacks, such as the predictions not always being up to date. Since there is a time lag on predictions, it might cost business. For example, let's say a manufacturing plant uses order forecasting models to acquire raw materials. Depending on the time lag of the batch forecast models, the business might need to bear the cost of the shortage in raw materials versus overstocking raw materials in the warehouse.

- **Online/transactional models**: Online models follow the pull paradigm; the prediction will be generated on demand. Online models take advantage of the current reality and use that for predictions. Online models are transactional, need low-latency serving, and should scale based on incoming traffic. A typical online model is a recommendation model, which could be product recommendation, design recommendation, and so on.

Though real-time prediction sounds fancy, online models face a different set of challenges. It is easier to build applications whose latency is 8 hours than it is to build an application with a latency of 100 milliseconds. The latency of online models is usually in milliseconds. That means the model has a few milliseconds to figure out what the most up-to-date value is (which means generating or getting the latest features for the model) and predict the outcomes. For this to happen, the model needs a supporting infrastructure to serve the data required for prediction. Online models are usually hosted as REST API endpoints, which again need scaling, monitoring, and more.

Now that we understand the difference between batch and online models, let's look at how batch model pipelines work.

Batch model pipeline

As discussed, batch model pipeline latency requirements can range from minutes to hours. Batch models usually run on a schedule, hence they will be orchestrated using tools such as Airflow or AWS Step Functions. Let's look at a typical batch model pipeline and how features are brought to production.

Figure 2.1 depicts typical batch model pipelines:

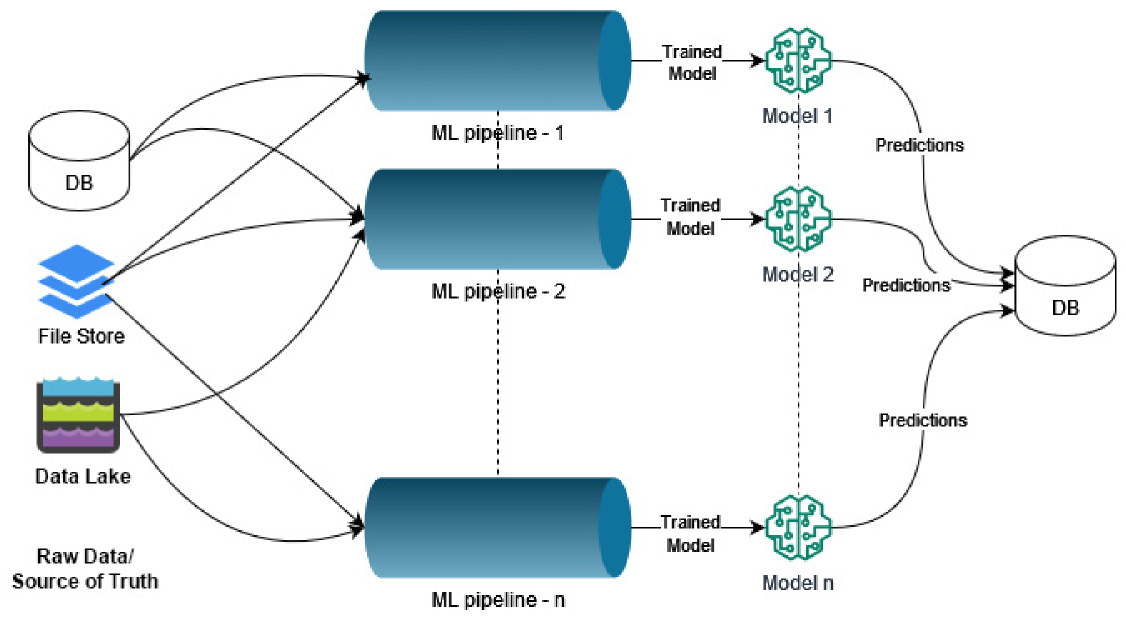


Figure 2.1 – Batch model pipelines

As discussed in ***Chapter 1***, *An Overview of the Machine Learning Life Cycle*, once the model development is complete and it's ready for productionization, the notebook will be refactored to remove unwanted code. Some data engineers also break down a single notebook into multiple logical steps, such as feature engineering, model training, and model prediction. The refactored notebooks or refactored Python scripts generated from the notebook are scheduled using an orchestration framework such as Airflow. In a typical pipeline, the first stage will read the raw data from different data sources, perform data cleaning, and carry out feature engineering, which will be used by subsequent stages in the pipeline. Once the model prediction stage is complete, the prediction output will be written to a persistent store, maybe a database or an S3 bucket. The results will be accessed from the persistent storage as and when needed. If a stage in the pipeline fails for any reason (such as a data accessibility issue or errors in the code), the pipeline will be set up to trigger an alarm and stop further execution.

If you haven't already noticed, in the batch model pipeline, the features are generated when the pipeline runs. In some cases, it also retrains a new model with the latest data, and in others, it uses a previously trained model and predicts using the features generated from the data available at the time when the pipeline is run. As you can see in *Figure 2.1*, every new model that is built starts at the raw data source, repeats the same steps, and joins the production pipeline list. We will discuss the problems in this approach in the later section. Let's look at the different ways to bring features to production in online models next.

Online model pipeline

Online models have the special requirement of serving features in near real time, as these models are customer-facing or need to make business decisions in real time. There are different ways to bring features to production in an online model. Let's discuss them one by one in this section. One thing to keep in mind is that these approaches are not exactly how everybody does it; they are merely a representation of group approaches. Different teams use different versions of these approaches.

Packaging features along with models

To deploy an online model, it will have to be packaged first. Again, there are different standards that teams follow depending on the tools they use. Some might use packing libraries such as MLflow, joblib, or ONNX. Others might package the model directly as the REST API Docker image. As data scientists and data engineers have a different set of skills, as mentioned in *Figure 1.1* of ***Chapter 1***, *An Overview of the Machine Learning Life Cycle*, the ideal approach is to provide data scientists with tools to package models using libraries such as MLflow, joblib, and ONNX, and save the model to a model registry. The data engineers can then use the registered model to build REST APIs and deploy it. There is also out-of-the-box support to deploy MLflow-packaged models as AWS SageMaker endpoints with a simple **command-line interface** (**CLI**) command. It also supports building REST API Docker images with a CLI command, which then can be deployed in any container environment.

While libraries such as MLflow and joblib provide a way to package Python objects, they also support adding additional dependencies if required. For example, MLflow provides a set of built-in flavors to support the packaging of models using ML libraries such as scikit-learn, PyTorch, Keras, and TensorFlow. It adds all the required dependencies for the ML library. Packaging models with built-in flavors is as simple as using the following code:

```
mlflow.<MLlib>.save_model(model_object)

## example scikit-learn

mlflow.sklearn.save_model(model_object)
```

Along with the required dependencies, you can package the **`features.csv`** file and load it in the **`predict`** method of the model. Though this might sound like an easy deployment option, the result of this is not far away from a batch model. Since features are packaged along with the model, they are static. Any change in the raw dataset will not affect the model unless a new version of the model is built with a new set of features generated from the latest data and packaged along with the model. However, this might be a good first step from batch to online models. The reason I say this is that instead of running it as a batch model, you have now made it a pull-based inference. Also, you have defined a REST endpoint input and output format for the consumer of the model. The only pending step is to get the latest features to the model instead of static features, which are packaged. Once that is implemented, the consumers of the model won't have to make any changes and consumers will be served with predictions using the latest available data.

Push-based inference

Unlike pull-based inference, where the model is scored when needed, in the push-based inference pattern, predictions are run proactively and kept ready in a transactional database or key-value store so that they can be served at low latency when the request comes. Let's look at a typical architecture of online models using push-based inference:

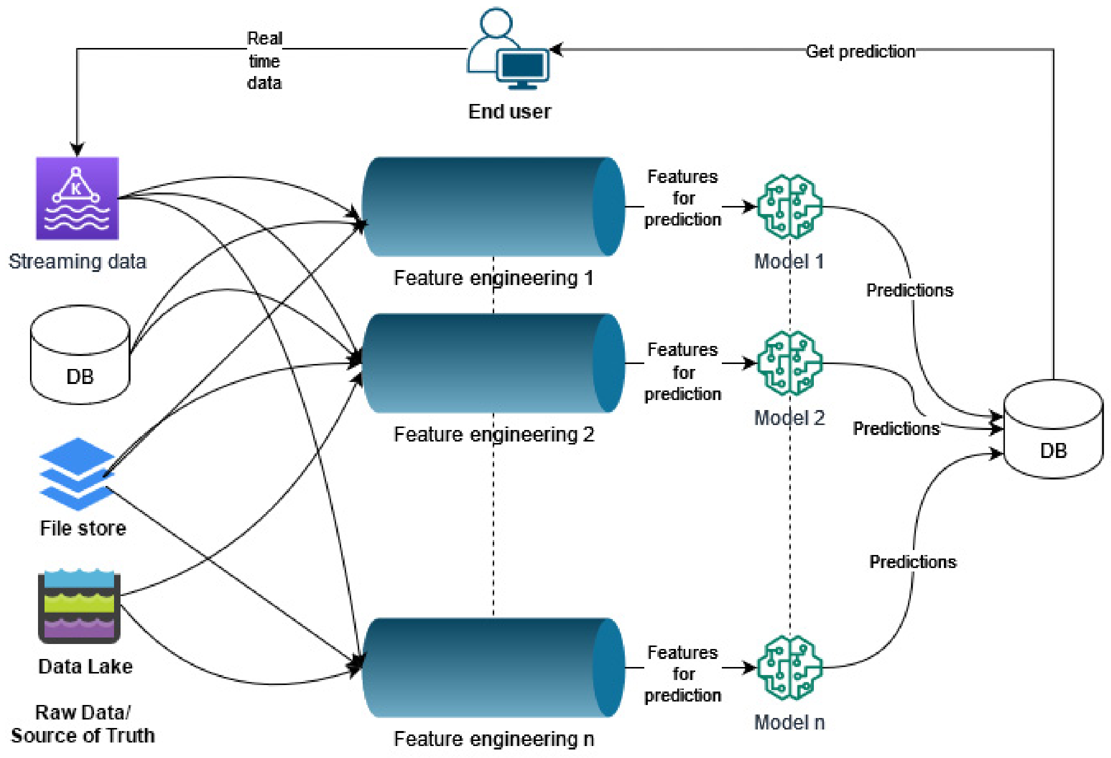


Figure 2.2 – Push-based inference

Figure 2.2 shows the architecture of push-based inference. The idea here is similar to batch model inference, but the difference is that the pipeline also considers the real-time dataset, which is changing dynamically. The operation of a push-based model works as follows:

- The real-time data (in this example, user interactions with the website) will be captured and pushed to a queue, such as Kafka, Kinesis, or Event Hubs.
- The feature engineering pipelines subscribe to a specific set of topics or a specific set of queues depending on what data is needed to generate features of the model. This also depends on the tools and the architecture. There may be a single queue or multiple queues depending on how huge/diverse the application is.
- Whenever an event of interest appears in the queue, the feature engineering pipeline will pick this event and regenerate the features of the model using other datasets.

  NOTE

  Not all features are dynamic. Some features may not change very much or very often. For example, a customer's geographic location might not change often.
- The newly generated features are used to run the prediction of the data point.
- The results are stored in a transactional database or a key-value store.
- Whenever required, the website or application will query the database to get new predictions for the specific ID (such as `CustomerId` when serving recommendations for a customer on a website).

- This process repeats every time a new event of interest appears on the queue.
- A new pipeline will be added for every new ML model.

This approach might seem easy and straightforward, as the only additional requirement here is real-time streaming data. However, this has limitations; the whole pipeline will have to run within milliseconds so that the recommendations are available before the application makes the next query for prediction. This is achievable but might involve a higher cost of operationalization because this is not just one pipeline: every pipeline for real-time models will have to have a similar latency requirement. Also, this will not be a copy-and-paste infrastructure because each model will have a different set of requirements when it comes to incoming traffic. For example, a model working on order features might require fewer processing instances, whereas a model working with clickstream data will require more data processing instances. Another thing to keep in mind is that although it might look like they are writing data to the same database, most of the time, it involves different databases and different technologies.

Let's look at a better solution next.

Pull-based inference

In contrast to push-based inference, in pull-based inference, predictions are run at the time of the request. Instead of storing the predictions, feature sets of a specific model are stored in a transactional database or key-value store. During prediction, the feature set can be accessed with low latency. Let's look at the typical architecture of a pull-based inference model and the components involved:

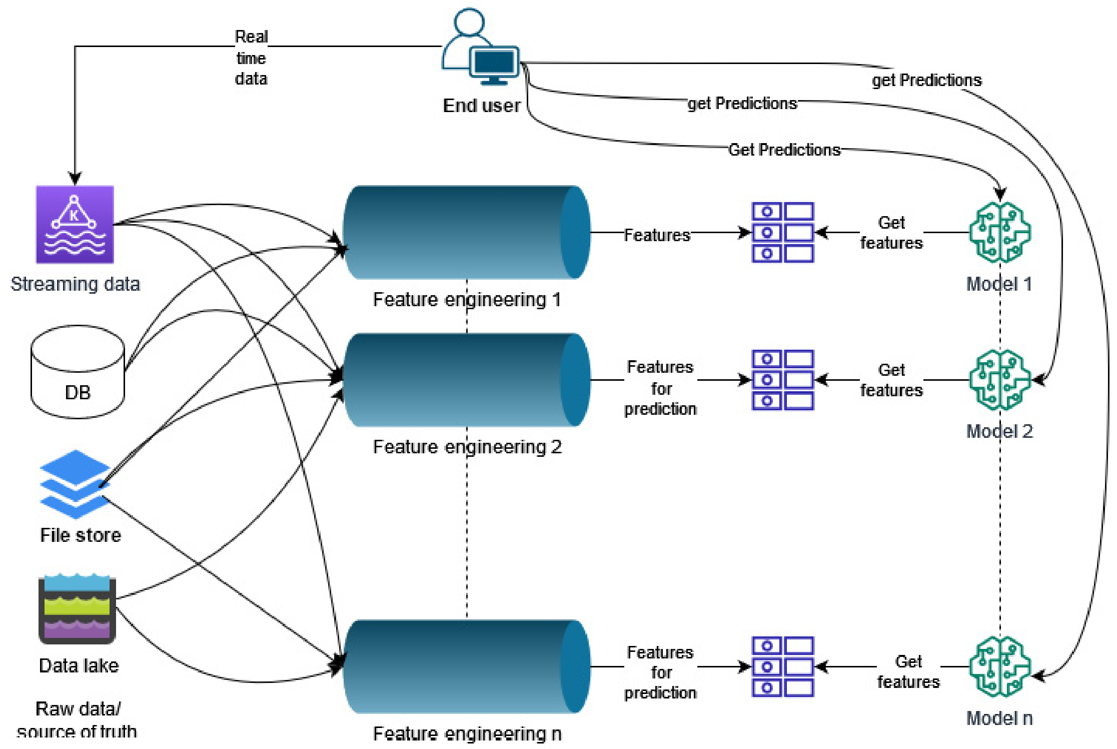


Figure 2.3 – Using transactional/key-value store for features

Figure 2.3 shows another way of bringing features to production: the pull-based mechanism. Half of the pipeline works in a similar way to the push-based inference that we just discussed. The difference here is that after feature engineering, the features are written to a transactional data-base or key-value store. These features will be kept up to date by the pipe-lines. Once the features are available, the model works as follows:

1. The model's **`predict`** API would have a contract similar to the one mentioned here:

   ```
   def predict(entity_id: str) -> dict
   ```

2. When the application needs to query the model, it will hit the REST endpoint with **`entity_id`**.
3. The model will use **`entity_id`** to query the key-value store to get the features required to score the model.
4. The features are used to score the model and return the predictions.

This approach is ideal if you don't have the feature store infrastructure. Also, we will discuss this extensively in the next chapter. Again, there are a few concerns involved with this approach, which are the repetition of the work, deploying and scaling the feature engineering pipeline, and managing multiple key-value store infrastructures, among others.

Calculating features on demand

Let's discuss one last approach before we move on and discuss the problems with these approaches. In the approaches discussed so far, the data pipelines calculate the features proactively as and when the data arrives, or when the pipeline runs. However, it is possible to calculate the features on demand when there is a request for the inference. This means when the application queries the model for a prediction, the model will request another system for the features. The system uses raw data from different sources and calculates the features on demand. This may be the most difficult architecture to implement, but I heard on the *TWIML AI* *podcast with Sam Charrington, episode 326: Metaflow, A Human-Centric Framework for Data Science with Ville Tuulos*, that Netflix has a system that can generate features on demand with a latency of seconds.

The **`predict`** API might look similar to the one in the last approach:

def predict(entity_id: str) -> dict

It then invokes the system to get features for the given entity, runs predictions using the features, and returns the results. As you can imagine, all of this will have to happen within a few seconds. The on-demand feature engineering to be executed in real time might require a huge infrastructure with multiple caches between different storage systems. Keeping these systems in sync is not an easy architecture design. It's just a dream infrastructure for most of us. I haven't seen one so far. Hopefully, we will get there soon.

In this section, we discussed multiple ways to bring features into production for inference. There may be many other ways of achieving this, but most solutions are variations that revolve around one of these approaches. Now that we understand why and how we bring features to production, let's look at the common issues that these approaches have and what can be done to overcome them.

Common problems with the approaches used for bringing

features to production

The approaches discussed in the previous section seem like good solutions. However, not only does every approach have its own technical difficulties, such as infrastructure sizing, keeping up with a service-level agreement (SLA), and interaction with different systems, but they have a few common problems as well. This is expected in a growing technical domain until it reaches a level of saturation. I want to dedicate this section to the common problems that exist in these approaches.

Re-inventing the wheel

One of the common problems in engineering is building something that already exists. The reasons for that could be many; for example, a person developing a solution may not know that it already exists, or the existing solution is inefficient, or there is a need for additional functionality. We have the same problem here.

In many organizations, data scientists work in a specific domain and with a team supporting them, which usually includes ML engineers, data engineers, and data analysts. Their goal is to get their model to production. Though the other teams working in parallel also have the goal of getting their model to production, they hardly ever collaborate with each other due to their schedules and delivery timelines. As discussed in the first chapter, every persona in the team has different skill sets, different levels of experience with the available tools, and different preferences. Also, data engineers on two different teams rarely have the same preference. This leads to each team finding a solution to productionizing their model, which involves building feature engineering pipelines, feature management, model management, and monitoring.

After coming up with a successful solution, even if the team (let's call it team-A) shares their knowledge and success with other teams, the response you would get would be *good to know*, *interesting*, *could be useful for us*. But it will never materialize into the other teams' solutions. The reason for that is not that other teams are indifferent to what team-A achieved. Apart from the knowledge, nothing that was built by team-A in many cases is reusable. The options that the other teams are left with are to copy the code and adapt it to their needs and hope it works or implement a similar-looking pipeline. Hence, most teams end up building their own solution for the model. The interesting thing is that even team-A will re-build the same pipeline for the next model they work on in most cases.

Feature re-calculation

Let's start with a question. Ask yourself this: *how much memory does your phone have?* Most probably you know the answer by heart. If you are not sure, you might check the memory in the settings and answer. Either way, if I or somebody else asks you the same question in an hour, I'm pretty sure you will not go back into the phone's settings and check again before answering, unless you have changed your phone. So why are we doing this in all our ML pipelines for features?

When you go back and look at the approaches discussed previously, all of them have this common problem. Let's say team-A just completed a customer LTV model successfully and took it to production. Now team-A has been assigned another project, which is to predict the next purchase day of the customer. There is a high chance that the features that were effective in the customer LTV model can be effective here as well. Though these features are being calculated periodically to support the production model, team-A will start again with the raw data, calculate these features from scratch, and use them for the model development. Not only that, but they replicate the whole pipeline, though there are overlaps.

As a result of this re-calculation, depending on the setup and the tools that team-A uses, they will be wasting compute, storage, and man-hours, whereas with better feature management, team-A could have gotten a head start on the new project, which is also a cost-effective solution.

Feature discoverability and sharing

As discussed, one of the problems is recalculation within the same team. The other part of this problem is even bigger. That is re-calculation across the teams and domains. Just like in the *Re-inventing the wheel* section, where teams were trying to figure out how to bring ML features to production, data scientists are re-discovering the data and features themselves here.

One of the major drivers of this is a lack of trust and discoverability. Let's talk about discoverability first. Whenever data scientists work on a model and do a great job of data mining, exploration, and feature engineering, there are very limited ways of sharing it, as we discussed in the first chapter. The data scientist can use emails and presentations to do that. However, there is no way for anybody to discover what's available and selectively build what is not and use both in the model. Even if it's possible to discover other data scientists work, it is not possible to use it without figuring out data access and re-calculating features.

The other driver for re-inventing the wheel in data discovery and feature engineering is trust. Though evidence is clear that there is a production model that uses the generated features, data scientists often find it difficult to trust programs developed by others that generate the features. Since raw data is trustworthy as it will have SLAs and schema validations, data scientists often end up re-discovering and generating the features.

Hence the solution required here is an application that can make the features generated by others discoverable, sharable, and, most importantly, trustworthy, that is, a person/team who owns and manages the features they produce.

Training vs Serving skew

One other common problem in ML is training and serving skew. This happens when the feature engineering code used to generate the features for model training is different from the code used to generate the features for model prediction/serving. This could happen for many reasons; for instance, during model training, the data scientist may have used PySpark for generating the features, where as while productionizing the pipeline, the ML/Data engineer who took over, used a different technologies that is required by the production infrastructure. There are few problems with this. One is, there are two versions of feature engineering code, and the other problem is this could cause the training versus serving skew since the data generated by two versions of the pipeline may not be same for the same raw data input.

Model reproducibility

Model reproducibility is one of the common issues to tackle in ML. I have heard the story of how after a data scientist quit his job, the model he was working on was lost and his team couldn't reproduce the model many times. One of the main reasons for this is again the lack of feature management tools. You might ask what the problem is in reproducing the same model when you have the history of raw data. Let's take a look.

Let's say there was a data scientist, Ram, working on an ML model to recommend products to customers. Ram spent a month working on it and came up with a brilliant model. With the help of data engineers on the team, the model was deployed to production. But Ram was not challenged enough in this job, so he quit and moved on to a different firm. Unfortunately, the production system went down, Ram didn't follow the

MLOps standards of saving the model to a registry, so the model was lost and could not be recovered.

Now, the responsibility for rebuilding the model is given to Dee, a new data scientist on the team, who is smart and uses the same dataset that was used by Ram, and performs the same data cleansing and feature engineering as if Dee is a reincarnation of Ram. Unfortunately, Dee's model cannot get the same results as Ram's. No matter how many times Dee tries, she cannot reproduce the model.

One of the reasons for this is that the data has changed over time, which in turn affects the feature values and hence the model. There is no way to go back in time to produce the same features that were used the first time. As model reproducibility/repeatability is one of the crucial aspects of ML, we need to time travel. This means that data scientists should be able to go back in time and fetch the feature from a specific time in the past, just like in *Avengers: Endgame*, so that the models can be reproduced consistently.

Low latency

One other problem that all the approaches are trying to solve is low-latency feature serving. The ability to provide features at low latency decides whether a model can be hosted as an online model or a batch model. This has problems such as building and managing infrastructure and keeping features up to date. As it doesn't make sense to set up all models to be transactional, at the same time there is a high chance that a feature used in one batch model could be of great use in a different online model. So, the ability to switch the low-latency serving on and off would be a great benefit to data scientists.

So far in this section, we have been through some of the common problems with the approaches discussed in the previous section. The question that still remains unanswered is what can be done to make this better? Is there a single tool or set of tools that exists today that can help us solve these common problems? As it turns out, the answer is *yes*, there is one tool that can solve all the problems we have talked about so far. It is called a *feature store*. In the next section, let's see what feature stores are, how they solve these problems, and the philosophy behind them.

Feature stores to the rescue

Let's begin this section with the definition of feature stores. A **feature store** is an operational data system for managing and serving ML features to models in production. It can serve feature data to models from a low-latency online store (for real-time prediction) or from an offline store (for scale-out batch scoring or model training). As the definition points out, it's a whole package that helps you create and manage ML features, and accelerate the operationalization of models. Before we dive deeper into feature stores, let's look at how the architecture of ML pipelines changes with the introduction of a feature store:

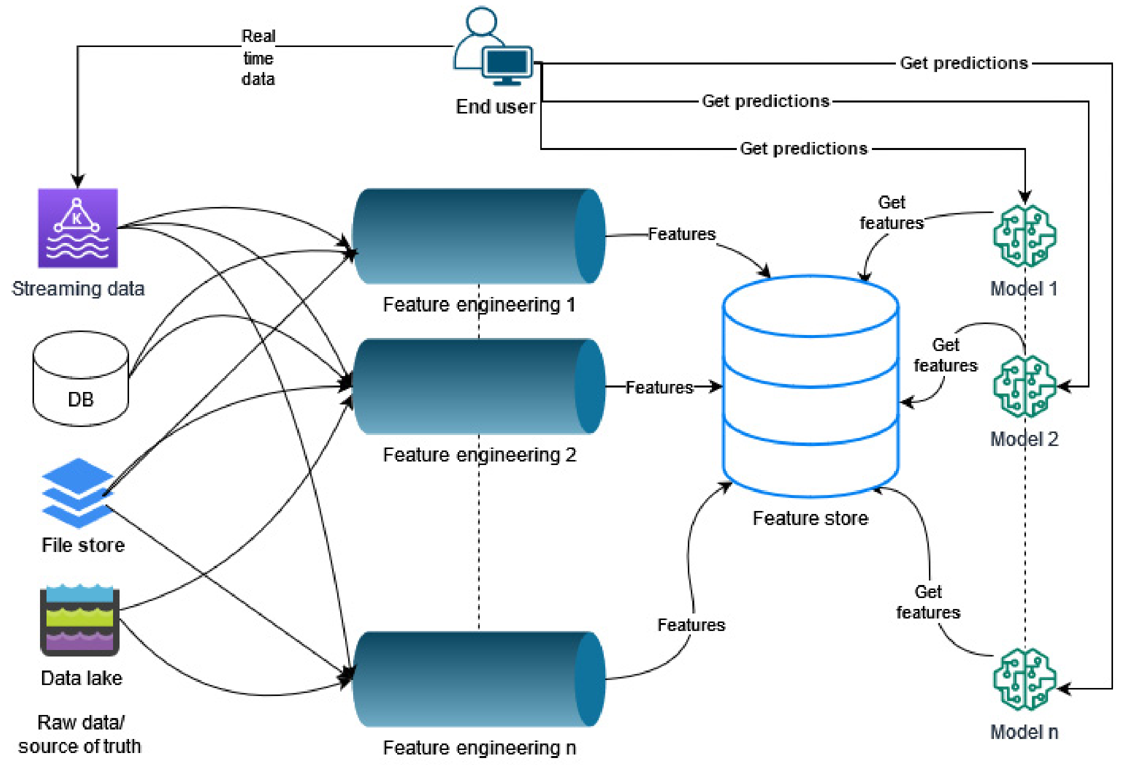


Figure 2.4 – ML pipelines with a feature store

Figure 2.4 depicts the architecture of ML pipelines when you include a feature store. You may think that *Figure 2.4* looks the same as *Figure 2.3* and I just replaced a bunch of small data stores with a bigger one and called it a feature store. Yes, it might seem that way, but there is more to it. Unlike a traditional data store, a feature store has a special set of capabilities; it is not a data store, but rather a data system (as its definition states), and it can do much more than just storage and retrieval.

With the feature store being part of the ML pipeline, this is how the entire pipeline works:

1. Once the data scientist has a problem statement, the starting point will not be raw data anymore. It will be the feature store.
2. The data scientist will connect to the feature store, browse through the repository, and use the features of interest.
3. If this is the first model, the feature store might be empty. From here, the data scientist will go into the discovery phase, figure out the dataset, build a feature engineering pipeline, and ingest the features into the feature store. The feature store decouples feature engineering pipelines from the rest of the stages in ML.
4. If the feature store is not empty but there are not enough features available in the feature store, the data scientist will discover the data of interest and another feature engineering pipeline will be added to sink a new set of features into the feature store. This approach makes the features available for the model that the data scientist is working on and for other data scientists who find these features useful in their model.
5. Once the data scientist is happy with the feature set, the model will be trained, validated, and tested. If the model performance is not good, the data scientist will go back to discover new data and features.
6. When the model is ready to be deployed, the **`predict`** method of the model will include code to fetch the required features for generating model predictions.
7. The ready model will be deployed as a REST endpoint if it's an online model; otherwise, it will be used to perform batch predictions.

Now that we understand how the pipeline works, let's go through the problems we discussed in the previous section and learn how the feature store solves them.

Standardizing ML with a feature store

Once the feature store is standardized at the team level, though there may be different ways of reading data and building feature engineering pipelines, but beyond feature engineering, the rest of the pipeline becomes a standard implementation. The ML engineers and data scientists don't have to come up with new ways to bring features to production. After feature engineering, the data scientists and ML engineers will ingest the features into the feature store. The feature store, by definition, can serve features at low latency. All ML engineers will have to do after that is update their `predict` method to fetch the required features from the feature store and return the predictions. This not only makes the life of the ML engineers easy, it sometimes also offloads managing feature management infrastructure.

Feature store avoids reprocessing data

As mentioned in the definition, a feature store has an offline store, and data from the offline store can be retrieved for model training or batch inference. Model training here doesn't mean training the same model. The features ingested into the feature store can be used for the training of another model.

Let's take the same example we used while discussing the problem: team-A just completed the production deployment of the customer LTV model. The next model team-A starts working on is predicting the next purchase date. When the data scientists start working on this model, they don't have to go back to raw data and re-calculate the features that were used to build the customer LTV model. The data scientist can connect to the feature store, which is being updated with the latest features of the previous model, and get the features required to train the new model. However, the data scientist will have to build a data cleaning and feature engineering pipeline for the additional features that they find useful from the raw data. Again, the newly added features can be reused in the next model. This makes model development efficient and cost-effective.

Features are discoverable and sharable with the feature store

In the previous paragraph, we discussed the reuse of features within a team. Feature stores help data scientists achieve that. The other major issue was re-calculating and re-discovering useful data and features across teams due to the lack of feature discoverability. Guess what? Feature stores solve that too. Data scientists can connect to a feature store and browse through the existing feature tables and schemas. If they find any of the existing features useful, data scientists can use them in the model without re-discovering or re-calculating them.

Another problem involved in sharing was trust. Although feature stores don't solve this completely, they address it to some extent. Since the feature tables are created and managed by a team, the data scientists can always reach out to the owners to get access and also discuss other aspects, such as SLAs and monitoring. If you haven't noticed yet, feature stores facilitate collaboration between teams. This can be beneficial for both teams, with data scientists and ML engineers from different teams working together and sharing each other's expertise.

No more training versus serving skew

With Feature Store, training versus serving skew will never occur. Once the feature engineering is complete, the features will be ingested into the feature store and feature store is the source for model training. Hence, the data scientists will use the features in feature store to train the ML model. Once the model is trained and moved to production, the production model will fetch the online store or historical store again for model prediction. As these features are used by both - training and prediction, serving is generated by the same pipeline/code and we will never have this issue with feature store.

Model reproducibility with feature stores

The other major issue with the previously discussed architecture was model reproducibility. This is an issue: the data is changing frequently, which in turn leads to features changing and hence the model changing, though the same set of features is being used to build the model. The only way to solve this problem was to go back in time and fetch the same state data that produced the old model. That may be a very complex problem to solve since it will involve multiple data stores. However, it is possible to store generated features in such a way that it allows data scientists to time travel.

Yes, that is exactly what a feature store does. A feature store has an offline store, which stores historical data and allows users to go back in time and get the value of a feature at a specific point in time. With a feature store, a data scientist can get features from a specific time in history, so reproducing the model consistently is possible. Model reproducibility is no longer an issue with feature stores.

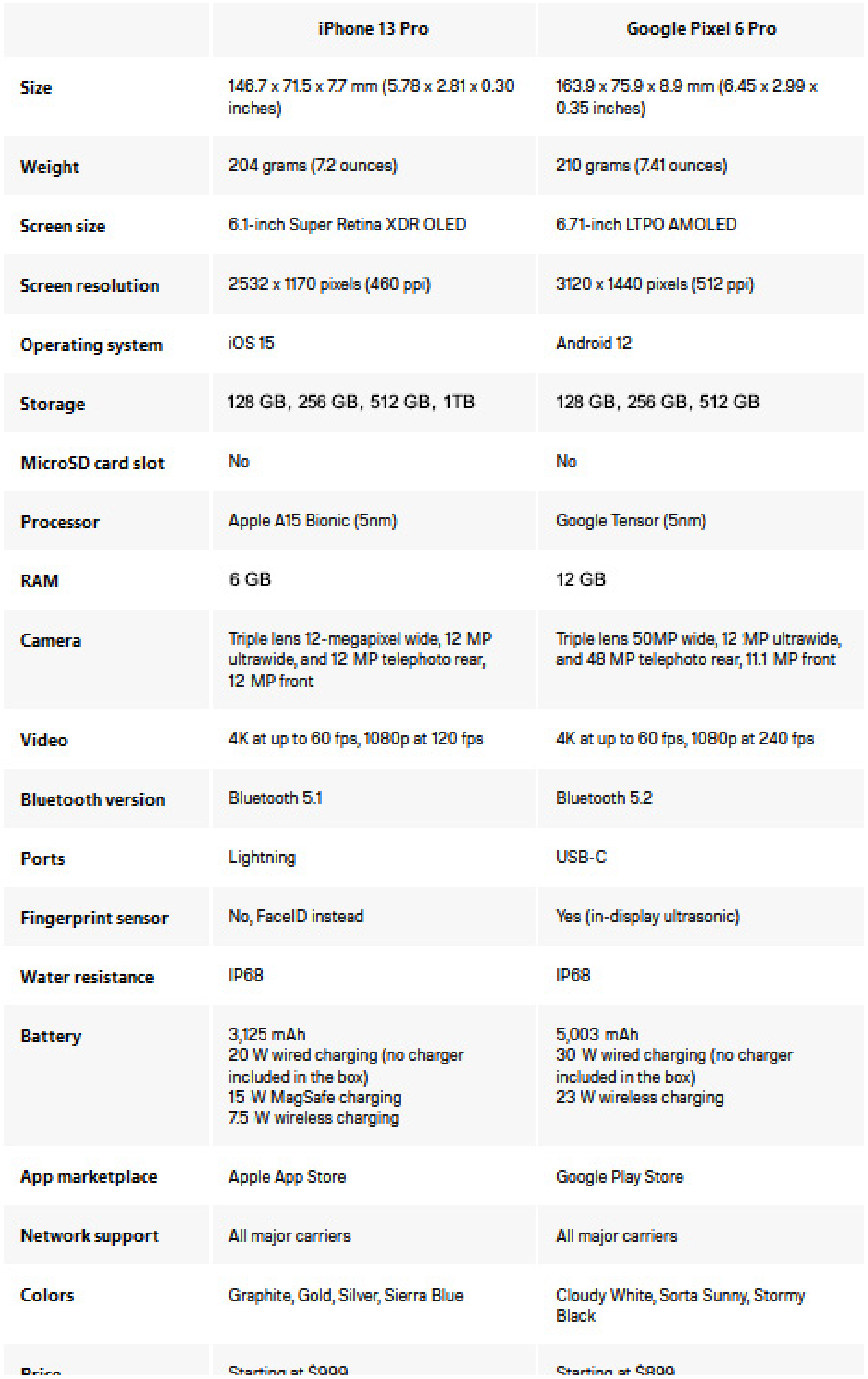
Serving features at low latency with feature stores

Though all the solutions were able to achieve the low-latency serving in some way, the solutions were not uniform. ML engineers had to come up with a solution to solve this issue and also build and manage the infrastructure. However, having a feature store in the ML pipeline makes this simple and also offloads the infrastructure management to the other team in cases where the platform team manages the feature store. Even without that, having the ability to run a few commands and have the low-latency serving up and running is a handy tool for ML engineers.

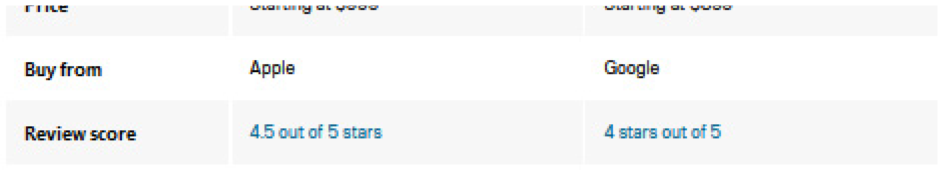
Philosophy behind feature stores

In this chapter, we have discussed different issues with ML pipelines and how feature stores help data scientists solve them and accelerate ML development. In this section, let's try to understand the philosophy behind feature stores and try to make sense of why having a feature store in our ML pipeline may be the ideal way to accelerate ML. Let's start with a real-world example as we are trying to build real-world experience with ML. You will be given the names of two phones; your job is to figure out which one is better. The names are iPhone 13 Pro and Google Pixel 6 Pro. You have an infinite amount of time to find the answer; continue reading once you have the answer.

As Ralph Waldo Emerson said, *It's not the destination, it is the journey*. Whatever your answer may be, however long you took to arrive at it, let's look at how you might have arrived at it. Some of you might have got an answer right away, but if you haven't used either of these phones, you probably would have googled **`iPhone 13 Pro vs Google Pixel 6 Pro`**. You would have browsed through a few links, which would give you a comparison of the phones:



	iPhone 13 Pro	Google Pixel 6 Pro
Size	146.7 x 71.5 x 7.7 mm (5.78 x 2.81 x 0.30 inches)	163.9 x 75.9 x 8.9 mm (6.45 x 2.99 x 0.35 inches)
Weight	204 grams (7.2 ounces)	210 grams (7.41 ounces)
Screen size	6.1-inch Super Retina XDR OLED	6.71-inch LTPO AMOLED
Screen resolution	2532 x 1170 pixels (460 ppi)	3120 x 1440 pixels (512 ppi)
Operating system	iOS 15	Android 12
Storage	128 GB, 256 GB, 512 GB, 1TB	128 GB, 256 GB, 512 GB
MicroSD card slot	No	No
Processor	Apple A15 Bionic (5nm)	Google Tensor (5nm)
RAM	6 GB	12 GB
Camera	Triple lens 12-megapixel wide, 12 MP ultrawide, and 12 MP telephoto rear, 12 MP front	Triple lens 50MP wide, 12 MP ultrawide, and 48 MP telephoto rear, 11.1 MP front
Video	4K at up to 60 fps, 1080p at 120 fps	4K at up to 60 fps, 1080p at 240 fps
Bluetooth version	Bluetooth 5.1	Bluetooth 5.2
Ports	Lightning	USB-C
Fingerprint sensor	No, FaceID instead	Yes (in-display ultrasonic)
Water resistance	IP68	IP68
Battery	3,125 mAh 20 W wired charging (no charger included in the box) 15 W MagSafe charging 7.5 W wireless charging	5,003 mAh 30 W wired charging (no charger included in the box) 23 W wireless charging
App marketplace	Apple App Store	Google Play Store
Network support	All major carriers	All major carriers
Colors	Graphite, Gold, Silver, Sierra Blue	Cloudy White, Sorta Sunny, Stormy Black
Price	Starting at $999	Starting at $899



Price	[illegible]	[illegible]
Buy from	Apple	Google
Review score	4.5 out of 5 stars	4 stars out of 5

Figure 2.5 – iPhone 13 Pro versus Google Pixel 6 Pro

This was a great way of comparing two phones. Some of you might have done a lot more to arrive at the answer, but I'm sure none of us went and bought both phones, read through the specs provided by Apple and Google, used each of them for a month, and became experts with each of them before answering the question.

In this task, we were smart enough to use the expertise and work done by others. Although there are a lot of comparisons on the internet, we chose the one that works for us. Not only in this task, but in most tasks, from buying a phone to buying a home, we try to use expert opinion to make a decision. If you look at it in a certain way, these are features in our decision-making. Along with the expert's opinion, we also include our own constraints and features, such as budget, memory if it's a phone, the number of seats if it's a car, and the number of rooms if it's a house. We use a combination of these to decide and take action. In most cases, this approach works, and in some cases, we might do a lot more research and become experts too.

The use of feature stores in ML is an attempt to achieve something similar; it is like Google for data scientists. Instead of a generic search like Google, data scientists are looking for something specific, and are also sharing their expertise with other data scientists. If what is available in the feature store doesn't work for a data scientist, they will go to raw data, explore, understand, become experts in it, and come up with distinguishing features about a particular entity, which could be products, customers, and so on. This workflow of ML with feature stores will not only help data scientists use each other's expertise but also standardize and accelerate ML development.

Summary

In this chapter, we discussed common problems in ML feature management, different architectures of productionizing ML models, and ways to bring features to production. We also explored the issues involved with these approaches and how feature stores solve these issues by standardizing practices and providing additional features that a traditional data store does not.

Now that we understand what feature stores have to offer, in the next chapter, we'll get our hands dirty with feature stores and explore the terminology, features, typical architecture of a feature store, and much more.

Further reading

- *Feast documentation*: **https://docs.feast.dev/**