# Design A Machine Learning Platform

***Q: Are we building this mainly for a website and/or app that millions of users interact with daily?***

A: Yes, there's a website and a corresponding mobile version that millions of people visit every day.

***Q: I'm assuming that we'll be supporting dozens of teams, tens of thousands of features, and potentially hundreds of models. Is that a correct assumption?***

A: Yes, that's the right scale.

***Q: Do we have access to a data lake of all the clickstreams and logs required to create relevant features?***

A: Yes, there's an HDFS data lake.

***Q: Is the data that we need to process on the order of petabytes?***

A: Yes, it is.

***Q: Along with that sized data, is there also the need for thousands of daily data jobs to create the tens of thousands of features and labels?***

A: Yes, that's reasonable.

***Q: Does the platform need to support deep learning models?***

A: For now, we'll stick to basic models.

**1. Gathering System Requirements**

As with any ML design interview question, the first thing that we want to do is to gather requirements; we need to figure out what we need to build and what we don't need to build.

We're designing a machine learning platform which supports dozens of teams, tens of thousands of features, and hundreds of models.

To accomplish this we'll need to design 5 components:

- Managed Data
- Managed Models
- Managed Model Hosting

- Managed Experiments
- Managed Monitoring

We'll need to rely on systems which we won't touch on or design:

- HDFS data lake which contains ingested clickstreams and logs
- The UI which millions of users interact with daily

**2. Managed Data**

We need to:

- Provide a way to create, find, and share features and labels for machine learning
- Have close integration with HDFS data lake
- Scale to thousands of daily data jobs on petabytes of data which will create features and labels
- Provide a way create, view, and update data metadata
- Incorporate data quality monitoring
- Provide online and offline data access
- Enable custom data transforms specific to models

Data metadata refers to information about the data itself. This can include the data size, the number of rows and columns, who consumes the data, data description, and data schema. As well, metadata can include metrics about the quality of the data including how often the data is updated, the number of missing values and the variance of each column, and finally a total percentage of missing data across all of the columns.

Architecturally, we'll need a dedicated Feature Store which will support Hive queries to transform the relevant raw clickstream/log data ingested from the data lake into features and labels we can use for our models. The Feature Store Hive queries will also compute the values for the data metadata such as the number or rows and columns for a particular data set. These computed values will be used for data quality monitoring. As well, these hive jobs will join features and labels together to create training examples used to train models. These will be placed in a dedicated Examples Store. The Example Store can be an HDFS cluster or S3 and will be used during model exploration, training, and retraining. Along with the Examples Store, we'll need a way to serve features in a low-latency way such that predictions can be obtained for models when in production serving real-time traffic. This can be a distributed cache such as Redis or a low-latency database such as Cassandra. We'll call this Online Serving. This setup

ensures that the exact same features we train on will be available in production while serving inferences to users. We also need to provide access points to the Feature Store so users can create, view, and share features and labels. The access points will include programmatic access as well as a simplified UI experience. These access points will provide users the ability to set the frequency at which the features are updated. As well, within the data metadata, users must specify the features and label columns so that the appropriate routing can be done downstream for the Examples Store and the Online Serving of features. The API access point allows users to explore the data more in depth than what the UI allows. It also enables users to create more complex jobs than the simple joins and aggregations supported in the UI. It's important to note that the Example Store and Online Serving will more than likely not be in sync all of the time. This is because labels will be obtained later based the user behavior which resulted from the features and a model. As soon as those labels are obtained, they will be joined to the features and made available in the Examples Store but there will probably be more up-to-date features since then. Finally, since features and labels may need to be tweaked for some particular model, we need to provide the ability to transform the features within the Example Store and Online Serving. These transforms could include feature scaling, subselections of features, or filling in missing values. We'll call this the Transformer. A transform can be implemented as user-defined functions or within Docker containers which implement a 'transform' function. Since the transformer is associated per model, the specific transformation will be referenced in the model configuration which we'll cover later.

We'll orchestrate the daily jobs of transferring relevant raw data from the data lake to the Feature Store by using Airflow. The periodic Hive jobs which create features and labels from the raw data will also be managed with Airflow. Teams will access the Example Store and Online Serving with an HDFS location and an HTTP endpoint respectively. The Transformer can transform all of the features in the Example Store and Online Serving in batches. We can have manual and algorithmic systems in place to check that the same features and labels aren't being duplicated across different teams. This will reduce redundant storage and processing.

**3. Managed Models**

We need to have:

- Consistent representation of models
- Guaranteed Model Pipeline consistency between exploration and serving predictions in production
- Ways to explore data and model combinations
- Support for basic machine learning models

- Resources for training, validation, and evaluation

To create a consistent representation of models we'll use the concept of Model Pipelines. The Model Pipeline will contain stages. The primary stage will be model inference. This is the stage in which features are provided to the model in order to get a prediction. Another stage, before the model inference, will provide the method of deserializing the features obtained from the Feature Store into something that the model can actually use. As well, this stage will apply the features not present in the feature store such as country, website path, device, or search terms. These features are based on the session of the user and not kept in the Feature Store. The final stage of the Model Pipeline will execute after the inference is complete and will transform the output of the model into something that is usable. For instance, the model may output a softmax of probabilities and we need to transform that into an actual prediction such as a product ID or video ID.

Next, we need to guarantee consistency of the Model Pipeline between model exploration and when moving a Model Pipeline into production. We'll create a Model Repository which will store serialized Model Pipelines. The Model Repository will also manage Model Pipeline metadata which contains:

- Example Store location
- Feature Store ID
- Team Owner and Contact Info
- Environment Details
- Online Serving Endpoint
- Transformer ID
- Training configuration
- Model ID, description, and performance

As well as Model Pipeline serialization, the Model Repository will provide libraries for deserialization so that the Model Pipelines can be loaded for hosting in production so we can serve predictions. We'll go over hosting in more detail later.

Now that we can represent and store models, we need to provide a way for teams to explore data and model combinations. We're going to offer two ways to do this. First, we'll host a simplified point-and-click UI which allows users to select Example Store data and apply it to an assortment of pre-defined models. As a step further, users will also have the option of using AutoML which will perform feature selection, model selection, and hyperparameter without

anymore user input. For more control over the modeling experience, we'll provide an advanced interface. This will include an API for the Feature Store allowing users to view and create features in terms of custom Hive queries. The newly created features will be available in the Example Store and Online Serving. As well, a library will be provided so users can create custom Model Pipelines. To take full advantage of the advanced interface, we'll want to provide a common workspace such as Jupyter Notebooks through a JupyterHub server managed by the platform. This will support user-level environments and container images as well as data governance.

Next, we'll need to provide teams the ability to incorporate the following machine learning models into their Model Pipelines:

- Linear/Logistic Regression
- Gradient Boosted Trees
- Random Forests
- K-means

As well, we need to support both gridsearch and Bayesian hyperparameter tuning. Lastly, we want to provide teams access to the compute resources required for training, validation, and evaluation.

Architecturally, we'll need a dedicated Model Repository can be a implemented with a NoSQL database such as DynamoDB. We also need to provide access points to the Model Repository so teams can create and view Model Pipelines. The access points will include programmatic access as well as a simplified UI experience. As well, we'll need a JupyterHub server which create Jupyter notebook servers for each user. These notebooks will need to interact with the Feature Store, Example Store, and Model Repository. Finally, we'll need a Training Cluster which provides the compute resources for training, validating, and evaluating Model Pipelines. This cluster will be a Hadoop cluster using YARN and Spark.

**4. Managed Model Hosting**

To serve predictions to users in production we need to provide a way to host a team's model. We'll need to support:

- Online model hosting
- Batch inferences
- Model retraining with most recent features and labels
- Potentially tens of thousands of requests per second

Online model hosting will enable teams to call an HTTP endpoint to obtain a prediction with low-latency. Batch inferences will enable teams to perform inferences across entire sets of users or products. One example where batch inference makes sense is if a client needed to generate personalized email templates for all users. It wouldn't be practical for that client to call the HTTP endpoint hundreds of millions of times. However, batch inferences can also be useful in cases where clients have strict latency requirements. Since batch inferences pre-compute all of the inferences offline before hand, we can eliminate the latency incurred when executing the Model Pipeline at runtime. We also need to support model retraining to avoid models becoming stale. This happens when the current model parameters are not optimized with respect to the most current features and labels being provided by Online Serving.

Architecturally, we'll need a dedicated Inference Service which can be implemented with several Docker containers behind a load balancer. Upon startup, the containers will call the Model Repository and the load the appropriate models. The container environments will match the environments used during model exploration. For online model hosting, clients will call the Inference Service and provide some identifier for which features to retrieve from Online Serving (such as a user ID). The retrieved features will then be run through the Model Pipeline specified in the request. The endpoint will return the result of the Model Pipeline. In the case that a feature is not available in the Feature Store, such as a the device type, the request must contain the feature. Deployments will be initiated through the Model Repository interface. When a request to promote a Model Pipeline to production, the Deployment Configuration service will be responsible for creating containers with the correct environments. It will then sanity test that the requested Model Pipeline can be executed on the container. As well, it will ensure that the the Online Serving endpoint specified in the Model Pipeline metadata is up and running. This ensures synchronization of the deployment of the Model Pipeline, the container environment in the Inference Service, and the online feature store. Separately, batch inferences will be created with Spark on a Hadoop cluster. The batch inferences will be made available in two ways.

- A distributed file system such as HDFS or S3 for scenarios which are not latency sensitive, such as generating personalized email templates
- An HTTP endpoint, just as online model hosting, for scenarios with strict latency requirements

Model retraining and batch inference calculations can be managed with Airflow.

**5. Managed Experiments**

To enable teams to experiment with different models, we need to cover 4 things:

- Support experimentation creation

- Support viewing in progress and historical experiments and their results
- Enable frequentist/Bayesian A/B testing as well as multi-armed bandits
- Support experiment stoppage within minutes

Experiments will be created in the Model Repository interface. Past and in-progress experiments can be viewed along with their results in the Model Repository interface as well. Finally, this is also where experiments can be stopped.

Architecturally, we'll need a dedicated Experiment Manager service. This service will be called by the Inference Service when a client requests an inference along with an experiment ID. The Inference Service will call the experiment manager with a user ID and experiment ID to get the treatment allocation for that user in that particular experiment. This treatment will indicate which Model Pipeline should be used when generating a prediction for a user. Now to obtain experiment results, we need to know how the users behaved when given a particular treatment. Based on the users' behavior for each treatment, we'll be able to make a decision on whether to launch a treatment or to keep the current experience in place. To get the user behavior we need the same resources that Managed Monitoring will require. Let's move on to Managed Monitoring and we'll see how they relate.

**6. Managed Monitoring**

To enable teams to monitor the results of their models that are in production, we need to provide 3 things:

- Interface to see history of model outputs
- Interface to see how customers have responded to the model outputs
- Drift Detection

Architecturally, we'll need a dedicated Monitoring Service. This service will be asynchronously called by the Inference Service with the predictions that the Inference Service has been making per Model Pipeline. To see how users behave given a particular prediction, we'll need a access to the data lake which contains the clickstream and service logs. This raw data will include the action a customer took of the customers given some prediction. To relate a prediction with a user response, we can supply a unique identifier for each prediction generated which can then be associated with a user's session. Both the Experiment Manager and the Monitoring Service will use a similar Spark or Hive job to join the user session actions with the predictions generated for that user. Given this joined data, the Experiment Manager will be able to calculate p-values in the case of frequentist A/B testing or update the Beta distributions in the case of multi-armed bandits. The Monitoring Service will be able to display to teams the progress or

regressions that the model is having on any metric of interest including business metrics as well as model metrics.

**7. Bonus #1**

Since we have time, how would you incorporate features that need to be updated more frequently than once a day? For instance, let's say we want a feature representing the number of units we have left in stock for a particular product.

Generally, the idea is to use design a streaming architecture for those features such that we can have near real-time values. For our architecture, we can use Spark Streaming. That means that when something comes in from Kafka, Spark Streaming immediately processes it with our specified transformations and then it sends it off to a destination. One of the destinations in our case would be Online Serving. It would update the feature value immediately. Online Serving would also need to make sure the value doesn't get overwritten by an older value when the Feature store attempts to update Online Serving with a potentially less recent value. We could so enforce this by checking timestamps during the ingestion from Feature Store to Online Serving. The other destination would be the Examples Store. A similar process would be followed.

**8. Bonus #2**

What if we wanted to add deep learning capabilities to this platform?

We would add whatever libraries we want to support (Tensorflow, PyTorch, or MXNet) to the Jupyter Notebook container images by default. We would also ensure those packages are included in the Training Cluster as well as the Inference Service. For the Training Cluster it will likely be beneficial to add support for training models on GPUs or even multiple GPUs. In the case that we have very large neural networks, we should consider using parameter server topology to implement large-scale distributed training. The Inference Service's online model hosting will likely still use CPUs but will benefit from a larger amount of RAM in the case of large models. Batch inferences will benefit from GPUs.

## 9. System Diagram