Gallery: A Machine Learning Model Management System at Uber

Chong Sun
chong@uber.com
Uber Technologies Inc.
San Francisco, California

Nader Azari
nazari@uber.com
Uber Technologies Inc.
San Francisco, California

Chintan Turakhia
chintan@uber.com
Uber Technologies Inc.
San Francisco, California

ABSTRACT

Machine learning is critical to the success of many products across application domains. At Uber, we have a variety of machine learning applications including matching, pricing, recommendation, and personalization. As a result, we have a large number of machine learning models to manage in production. Generally, building machine learning models is an iterative process and machine learning models span across a set of stages of a lifecycle. In this paper, we describe Gallery, a machine learning model lifecycle management system to save and serve models and metrics and automatically orchestrate the flow of models across different stages in the lifecycle. We then use the Uber Marketplace Forecasting and Simulation platforms as examples to show how Uber uses Gallery in production and the benefits we get by using Gallery.

1 INTRODUCTION

Machine learning is critical to the success of many products across application domains. Companies employ machine learning for recommendation, targeting, and personalization. Uber uses machine learning across product features including matching, pricing, personalization, ETA estimation, and Uber Eats recommendations. Recently, there have been various systems and frameworks [1, 12, 22, 26] designed and built to make machine learning easy-to-use and scalable in production systems. However, as the interaction of models with systems have become more complex, a growing technology need exists to manage machine learning models through their lifecycle to accelerate the process of getting a model from exploration to production and improve the model iteration velocity.

Building machine learning models is an iterative process [7]. Given a problem to solve, the common lifecycle of a model, as shown in Figure 1, starts with model exploration, during which we design and explore multiple models. When we find a model that beats a benchmark, we build the model into a production system. Getting a model into production starts at the model training, where we generate model instances. We refer to the trained models as the instances of a model. We evaluate the performance of trained model instances and deploy instances when the performance is above certain thresholds. Otherwise, we continue to improve the models. When models are deployed in production, monitoring performance is critical. If a performance degradation is detected or we have a new model, we will need to re-train the appropriate model, deprecate the old model instances, and deploy the new model instances.

Managing a handful of models is feasible for a production system. However, operational scale quickly becomes untenable

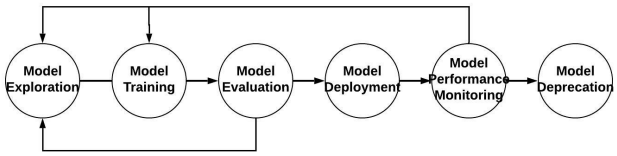

Figure 1: Machine Learning Model Lifecycle

with multiple machine learning problems, and more so when each problem has hundreds of model instances to manage. For example, when doing Marketplace-level forecasting at Uber, we forecast supply, demand, and other quantities in real time for hundreds of cities across the globe. We shard the problem spatially by city, training a model instance for each city-quantity combination because Uber is operating in many cities across the world, and different cities may pose different geospatial characteristics. Besides, the Uber business might be at different growth stages for different cities.

Though there are variances across applications and projects, many interesting questions about how to manage a large number of machine learning models in production are common. In this section, we list a sample of the questions which are raised between use cases: Where do we save and serve the models generated during model exploration or trained in production? How do we efficiently search for models and their experiment results? How can we confidently deploy a large-scale number of models and avoid regressions? How and when do we trigger model retraining due to model performance deterioration? In a complex system like the Uber Marketplace, the result of applying one model could be the input to another model. How can we manage the dependencies between multiple models?

How to address the above model management questions often depends on the experiences of machine learning engineers who work on these problems. Even within one company like Uber, different machine learning applications may use different approaches to solve these problems. For example, prior to Gallery there were over seven different storage solutions (e.g., MySQL, HDFS, Cassandra and Git repo) engineers used to save machine learning models. As a result, similar functionalities to manage machine learning models are scattered across a variety of production systems. This results in increased overhead to build and maintain individual systems, and causes a loss of visibility into the machine learning models across a production system. With the increasing number of machine learning solutions being built to solve business problems at Uber, we built Gallery, a model lifecycle management system to systematically and uniformally address these common questions to improve machine learning model velocity and productivity across Uber.

Gallery was started as a system to solve Uber Marketplace Forecasting model management problems and was later integrated as part of Michelangelo [6], Uber's ML Platform. It is a

system designed to manage machine learning models by providing functions for model saving, searching, serving, performance monitoring, and orchestration across different stages of the model development lifecycle.

Overall, the major contributions in this paper are as follows:

- A comprehensive analysis of the problems we addressed at Uber in order to manage machine learning models in a large-scale, distributed, microservices-based system.
- We describe Gallery, a model lifecycle management system used in production at Uber.
- We use Uber Marketplace Forecasting and Simulation case studies to demonstrate how we utilize Gallery to manage machine learning models and the benefits we have gained with Gallery.

2 THE MODEL MANAGEMENT PROBLEM

We first share some machine learning model management context at Uber before we define the problem. At Uber, we employ numerous machine learning applications, such as request dispatching, pricing, user growth, and recommendations. Overall, the Uber platform is microservice-based. Application teams build their own services and have different requirements for cadence and latency, implying different patterns of running the models. For example, long-term forecasting predicts hourly trips for a city for weeks in the future, while real-time forecasting predicts sub-hour demand. As a result, different applications might use different languages, modeling techniques, and frameworks for building and serving models.

In addition to the variety of applications and models, Uber is operating in markets across the world that have unique conditions in terms of population, city layout, climate, and population density. As a result, it is common to see machine learning models trained separately for different markets. We also need to frequently retrain the models when we detect model performance degradation due to the changing market conditions, and we need to independently trigger the retraining of the models for a city. Often it is not efficient to blindly re-train the models for all the cities, e.g., the training data for real-time demand forecasts can easily go up as much as terabytes for one city. Instead, we would like to retrain the models periodically if performance evaluation shows the need.

To solve the model management issue across heterogenous use cases, a system to manage a variety of machine learning models across different frameworks, languages, and usage patterns, from model exploration to models deployed in production, is necessary. To be clear, we refer to a machine learning model as an abstract data transformation which we can use to solve a particular problem or business use case. A model contains the specification of the input, output, and transformation, e.g., linear regression or random forest classification and all the corresponding hyperparameters. A model instance consists of a set of coefficients that is a learned representation of a given model on a particular training data set. The terms "model" and "model instance" are commonly used interchangeably when there is no ambiguity. Accordingly, we define the model management problem as: how to consistently and scalably manage a large number of complex models and model instances across stages of a model lifecycle.

3 THE GALLERY SYSTEM

In the section, we describe Gallery, a model lifecycle management system, built at Uber to solve the aforementioned model management problems. We first introduce the principles that guide our design. Then, we discuss the overall system architecture, followed by the description of each major system component.

3.1 Design Principles

Immutable. Any machine learning model and model instance generated and managed in our system is immutable. Any update of a model or model instance will result in a new version in production. This is critical to guarantee no unexpected behavior in production, and ensures that all decisions can be traced back to a specific model version. This builds the foundation for model performance observability and debuggability.

Model Neutral. Each model is treated as a black box and the model management system does not interpret the models. In this way, we can have one system to provide management for the varying models built for each application, e.g., a deep learning model using TensorFlow or PyTorch, or linear regression models using scikit-learn. Users are not blocked from leveraging the model management system because of their modeling technology choices.

Framework Agnostic. Any framework for model training, evaluation, deployment and serving, e.g., model exploration with Python code manually on a local server or scheduled pipelines to train models in production, could be seamlessly integrated with the model management system. With this flexibility, we can manage models from all different machine learning projects at different development stages. This lowers the on-boarding cost for new users and provides model management support for a wide array of use cases.

Automation. With a large number of machine learning models and model instances, automatically moving models across different stages in the lifecycle is the key to high scalability and velocity. Achieving automation requires the management system to have an integrated holistic view of the model workflow including training, evaluation and deployment.

3.2 Overall Architecture

We show the overall view of the Gallery architecture in Figure 2. Advanced model management is a core component of a machine learning system as it orchestrates the flow of a model across different stages of a lifecycle. For the sake of completeness, we include a generic machine learning system that encompass the basic stages of a machine learning lifecycle and the data infrastructure we leverage in Gallery for the storage in the architecture. We describe the major components of the Gallery system separately in the rest of this section.

3.3 Data Model

To manage the lifecycle of a machine learning model, Gallery collects data of *models*, *model instances*, and *model performance*, and the corresponding metadata information. We present a simplified version of the basic Gallery data model in Figure 3.

3.3.1 Model. A machine learning model is generally a representation of a transformation from a given input to a given output. We use model metadata to store the basic model information including the model owner, model description (e.g., linear

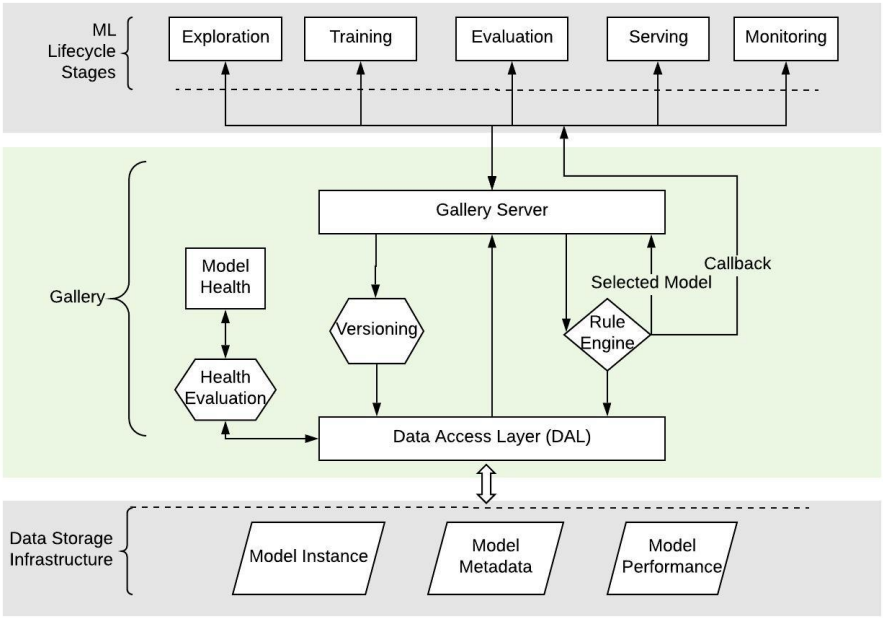

Figure 2: Gallery Overall Architecture

regression formula or neural network structure), features, hyperparameters, and also the information on how the model can be trained and served.

Building machine learning models is always an iterative process through which we generally start with a simple baseline model and subsequently improve the model performance by optimizing the model structure, tuning the hyperparameters, or updating the model features. Therefore, we keep track of the evolution of a model through next and previous pointers in the model record. In a complex production system, we often have one model depending on the output of other models. To get a holistic view of the application of machine learning models in such a system, we also keep track of the model dependency via upstream and downstream pointers.

3.3.2 Model Instance. A model instance is a realization of a model given a set of training data. It consists of the model parameters learnt from the training data and it is used to construct the model in serving for prediction. To achieve model neutrality, we treat model instances as uninterpreted binary blob data and any updates to the blob will be versioned as a new instance in Gallery. As a result, Gallery can not interpret any model and treats all the models the same. Depending on the types of models, the model instance sizes vary from a few KBs to 10s GBs. Model blob storage is abstracted from the users, and the blob is saved via Gallery in distributed data storage systems, e.g., S3 or HDFS.

We decouple the storage of the model instance blob with other model information. Each model instance has a field to record the model instance blob location, which could be a HDFS or S3 path.

For a model instance, we use metadata to keep track of the training data, training framework, and other configurations (e.g., seed for random number generator, number of epochs for training a deep learning model) we have set for the training to generate the model instance. Storing all the information about the models and model instances allow users of Gallery to closely reproduce their model instances on demand. Note that it is not always possible to generate exactly the same model instance due to the randomness introduced in training the models.

3.3.3 Model Performance. We track the performance of every model instance for offline model evaluation and online performance monitoring. When users measure their models either offline or online, they can write blobs of evaluation metrics that pertain to a specific mode instance. Each metric also has its own set of metadata to describe the nature of the evaluation. We store metrics as blobs in order to remain model neutral and framework agnostic. For different model evaluations, we can have different metrics, e.g., precision, recall, AUC for classification models and MSE (Mean Squared Error), MAPE (Mean Absolute Percentage Error) for regressions models. There are also a lot of customized metrics defined for application-specific evaluations. Gallery treats all the metrics the same and the metrics take the form of a structured blob with the basic format of "<metric>: <value>" pairs.

3.3.4 Metadata. As shown in the Figure 3, for Gallery models, model instances, and model performance, we keep a comprehensive set of metadata to identify model ownership, associate each model with its serving context, and link models to their training datasets. With metadata, we can improve the discoverability of models and instances by enabling search over key metadata fields. We record all necessary configurations, e.g., training code pointer, hyperparameters, and training data location and version, to make model instances reproducible. We provide a standard set of metadata fields and naming conventions to unify the characteristics of a model over a production system.

Note that none of the metadata about a model or a model instance is generated in Gallery. Users of Gallery need to save the information to Gallery via APIs within the Gallery server. An example of the usage of the Gallery APIs is presented in Section 4.1. Gallery simply manages the information and indexes the data

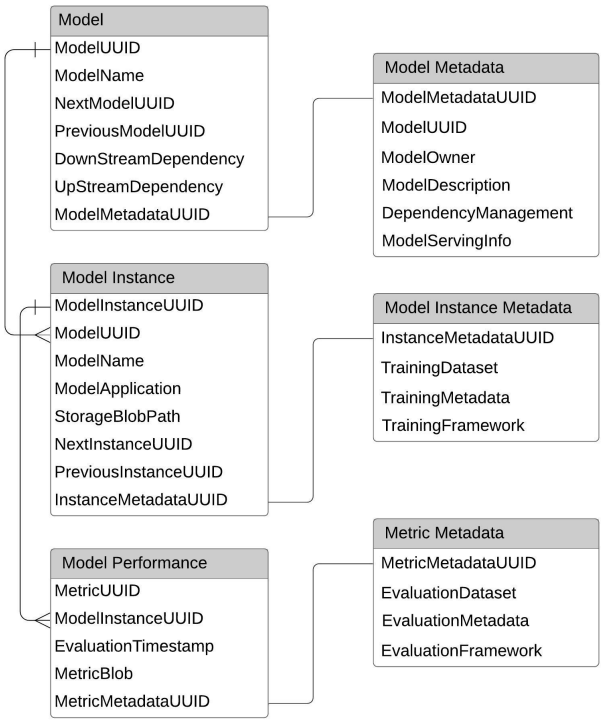

Figure 3: Gallery Basic Data Model

for querying. As a result, Gallery is model neutral and agnostic to any modeling framework.

3.4 Model Versioning

Model versioning is an approach to uniquely identify a model or model instance. It is the foundation for model immutability. With versioning, we never override an existing model or model instance. Any update to a model or model instance will introduce a new version. We keep track of the update history as the lineage of the model or model instance.

Versioning of code or artifacts is a basic requirement for any production system. While it is standard to use Git for code version control, there is no such standard for versioning models or model instances. As a result, many users derive their own versioning schemas, like Semantic Versioning [8] and timestamps, which lead to high maintenance costs due to lack of standandarization across users users and applications. Gallery abstracts model versioning away from users, analogous to what Git provides to code, and provides APIs for users to trace model lineages.

3.4.1 Model Instance Versioning. Each model can have one or multiple model instances. We not only version models, but also model instances. This is because both models and model instances have their own notions of change that need to be tracked. An update to a model represents some change to the underlying data transform such as feature and hyperparameter changes. Typically, these changes happen in response to new approaches for solving a problem and are usually less frequent than model instance updates. Model instance versions represent updates against an existing model with new training data. In production, periodic retraining is expected as new training data becomes available, and information about retraining is captured in the model instance versioning.

The versioning approach we took before Gallery is based on semantic versioning using the format of "<major>.<minor>.<patch>". A version example for a demand forecasting model instance is "1.3.10". We adhere to the following basic version updating rules: 1) update major versions when model architectures change, e.g., from linear regression to neural network; 2) update minor versions when features or hyper-parameters change, e.g., adding a new feature, and 3) update patch versions when the model instance is retrained. This approach works well when we have one simple forecasting model for a handful of cities. However, it is not manageable when we build and launch multiple forecasting model for hundreds of cities. As different models might perform better or worse for different cities and the forecasting model performance might degrade gradually due to the changes in the Uber business, we expect retraining models to improve model performance. However we do not want to retrain models for all the cities if one city performs poorly since that needlessly wastes computing resources. As a result, we very quickly end up with multiple model versions for different cites in the production system which becomes impossible to manually manage. The basic semantic versioning schema also loses meaning because cities are no longer aligned against the same versions.

In Gallery, instead of incorporating model semantics into the versions, we adopted a Git style versioning approach and assign a UUID for each model instance. We associate metadata to capture the model semantics and make it easy to search for. To be specific, when users create a new model, they declare a base version id for the model. The base version id is the top-level identifier that is linked to all its descendent model instances. Typically, a base version id represents some approach to solving a particular problem (e.g., demand forecasting). Each time a model instance is trained, a unique identifier is assigned to the trained model and its metadata tracks which base version id the instance was trained from. In this way, users can query for specific model instances, or traverse the evolution of their model by following all instances linked to a given base version id.

Figure 4 shows one example of a model and model instance. There are two base model version ids "demand_conversion" and "supply_cancellation" which represent models for the corresponding business problems. For example, "supply_cancellation" has evolved over four iterations with different model instances which are identified by four different UUIDs. The model instances are sorted by time and linked to the base model they were trained from.

3.4.2 Dependency Management With Versioning. Besides model and instance identification, versioning is at the core of model dependency tracking and management. As collaboration grows within a production system and models become more advanced, there are scenarios where models become dependent on one another. For example, the output of one demand forecasting model could be used as a feature for a pricing model. As systems become more complex, these types of relationships become very difficult to track. Tracking these relationships is an important prerequisite for understanding how a model impacts the entire production environment with a holistic view. Users need to be aware of the consequences of changes in their models, or need to be aware that changes in their model's behavior could be due to upstream dependencies. For example, the performance of Model A could improve even though the only change is on its upstream Model

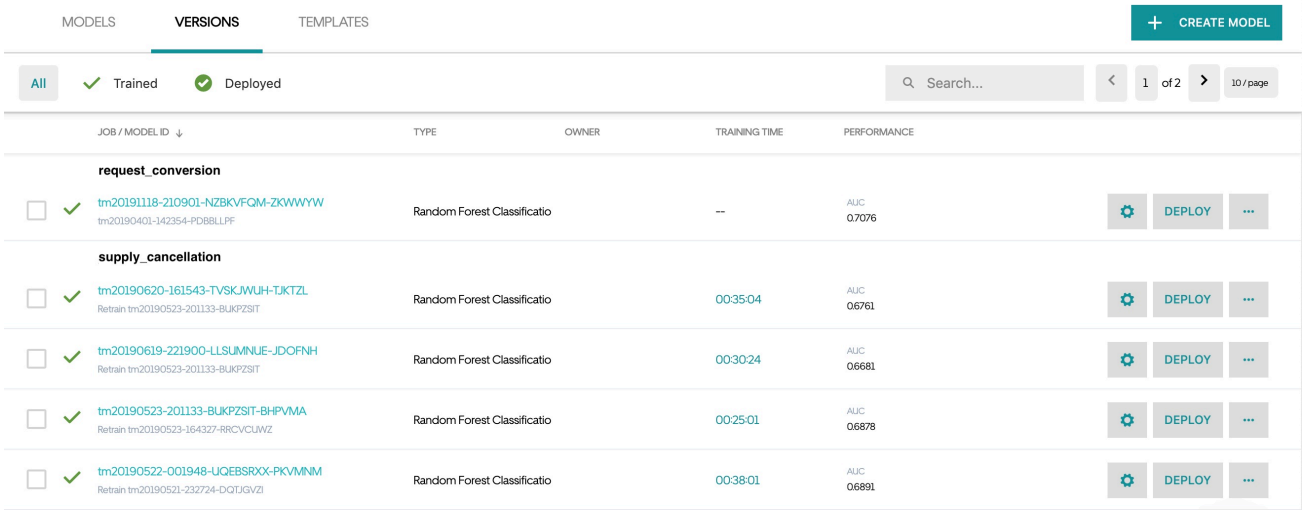

Figure 4: Model and Model Instance Versioning

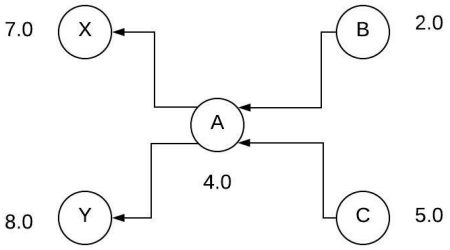

Figure 5: Model Dependency Graph

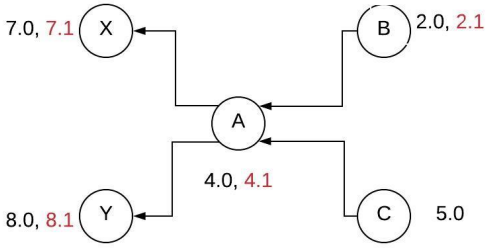

Figure 6: Model Dependency Update

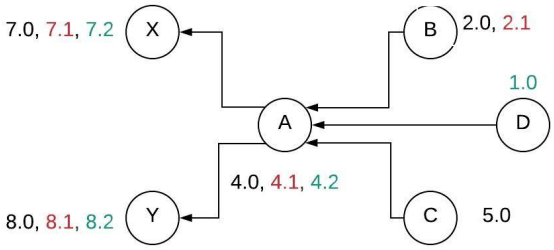

Figure 7: Adding New Model Dependency

B. Without tracking this dependency relationship, we would lose track of Model B's impact on the production system.

Here, we present one example to show how dependencies of 5 models are managed by Gallery. In Figure 5, we show a dependency graph of five models. Both Model X and Y depend on Model A and Model A depends on Model B and C. For readability, we use numbers instead of UUIDs to represent the model instance versions in this example. In Figure 6, we show the case of a model dependency update. When we update instance of Model B from version 2.0 to 2.1, this triggers the version updates for all Model B's downstream models including A, X and Y. Considering that there is no real change of Model A, X or Y, we automatically update the model instance version by adding a new model version to Gallery without changing the production versions. The owner of Model A can choose to upgrade to the new model version, if they want to include the updated Model B. But, models are not automatically updated because we would like users to be aware that their model dependencies have changed before their production environment is updated.

Figure 7 shows a use case when we add a new model dependency for an existing model. By adding the Model D as the dependency of Model A, we will automatically update model A's instance version to 4.2. Accordingly, the downstream Models X and Y will also be updated to instance version 7.2 and 8.2 separately.

Dependencies between models are established by the user when models are first registered in Gallery. When adding models, Gallery provides operations for the user to add dependent models by their uuids. There are also operations exposed for updating or removing dependencies. Once the dependencies are established, Gallery provides users with APIs to query their model's upstream or downstream dependencies and will track model updates across dependencies.

3.5 Model Storage

The Gallery model storage layer defines the interface through which model blobs, metadata, and metrics are stored and retrieved. We have the following model storage requirements: *searchability*, *agnostic*, *high availability*, and *low-latency*.

To satisfy these requirements, we build the Gallery model storage using a hybrid approach. Considering that model metadata

and metrics are commonly structured data, we use a relational database, e.g., MySQL, for storing metadata and providing support of flexible queries. The MySQL service is supported by the Uber infrastructure team to guarantee high availability and deployed cross data centers. Considering the enormity of models and model instances, we would not be able to scale up to handle thousands of models and instances if we need to interpret each model. As a result, we treat each model equally and store each model blob as binary data. We leverage Uber's large data storage service built on top of S3 and HDFS to store the model blobs to achieve model neutral and framework agnostic design principles. We expect Gallery users to provide their models as serialized binaries, which are in turn stored in Uber's large data storage service. The storage locations are subsequently stored as part of the model metadata so that they can be retrieved at serving time. Another benefit of taking this approach to save model blobs is that it does not have data size constraints, which can be an issue for large deep learning models. To handle cases of inconsistent data due to system failures, e.g., MySQL or HDFS write fails, we always write model blobs first and only write the model metadata after the model blobs are successfully stored. If the model blob of a model instance is saved but the metadata fails to save, then the model instance will not be available in the system.

Model metadata searchability is critical for users managing a high volume of models. Users conducting experiments or managing production environments need the ability to easily search and query models based on key metadata like training dates, model type, and features. Model searchability allows easy tracking of all models in the wild and more efficient analysis and experimentation over the various models.

Briefly, model storage is accessed via a unified DAL (data access layer). The model performance metrics are saved and read from MySQL to support flexible queries for analysis and monitoring. When models instance blobs are queried, the request first goes to MySQL to get the location of the model blob, and then the model is directly accessed via the storage location. The cache is updated with the requested blob and then is subsequently returned to the user.

3.6 Model Performance and Health

When building and maintaining production-grade software systems, it is standard to define SLAs with consumers to establish accountability and trust. Typical SLAs for software systems include availability, latency, and throughput. For machine learning systems, we also would like to have SLAs on performance. We define model health as a set of metrics and standards for users to adhere to in order to guarantee some level of accountability of models in Gallery.

More specifically, we define two categories of metrics to measure the model health. One category of the metrics is on the completeness of model information, which consists of metadata for model reproducibility and model performance. Production models should contain enough metadata to reproduce the model and annotate the behavior leading to a decision. Different models may have different performance metrics. In Gallery, we ensure that the performance of each model is evaluated and stored for monitoring and analysis.

The second category of model health metrics provides a holistic view of model performance across model lifecycle stages including training, validation and production. All model performance is agnostic to the system and provided by the applications. Model training performance is generally available as a by-product of model training. Model validation performance is produced by validation processes or backtesting and is used to check for model overfitting or as a gauge of whether to deploy a model to production. Model production performance is measured against served predictions and reflects the online performance of a model. The evaluation criteria for each performance metric is entirely up to the user and is configurable, since different models and applications optimize for varying outcomes. We store an object of metrics in Gallery and define the above metrics as guidelines for users.

With model performance metrics, we can derive various insights about the models in Gallery. The insights can give model owners a signal on how their model behaves over time, information about their serving environment, and establishes a level of trust between model owners and model consumers. Here are two examples of insights that Gallery can provide: model drift and production skew.

Model Drift . Model drift refers to the case when the statistical properties of the target variable, which the model is trying to predict, change over time in unpredictable ways. With real-time platforms, data changes. Accordingly, if the data we use in production gradually evolve to have different patterns from the data we use in the training, we may see the model performance degrade over time. Considering Uber's rapid growth in many markets, this drift can occur and once detected, triggers model re-training via Gallery rule engine using the new training data.

Production Skew. Production skew is the difference between performance at training time and serving time. Multiple factors can result in production skew, such as bugs in training or serving implementation, or discrepancies between training data and data feeding to model serving. The ability to detect production skew is critical for model performance monitoring.

3.7 Orchestration Rule Engine

As the number of models and model instances grow in a production system, it becomes increasingly difficult to manually manage their various states. Therefore, we designed and built a rule engine in Gallery to orchestrate the model workflows. Based on the model metadata, such as deployment configuration and various model performance metrics in Gallery, users can define conditions and actions in rules to automatically move the models across the stages of the lifecycle, such as model deployment and serving, monitoring and retraining, and deprecation.

In the following section, we first show the basic automations we need in the model lifecycle stages where the rule engine can help. Then, we describe our design of the rule engine system and illustrate how it works with examples.

Model Deployment and Serving Automation. When we generate model instances through training, we decide whether to deploy a model instance to production and replace the existing instance. It is common to have multiple models and instances deployed in production and use rules to select the best performer for serving, based on performance metrics generated by evaluation systems. Normally, different applications will have their own evaluation systems to measure the performance of the models. For example, in real-time forecasting, we have a heuristic model which uses the mean value of last 5 minutes as the forecasts. The heuristic model is stable and consistent, but may not always produce the best performance. We also have complex forecasting

models that take in more features, like historical data, into the prediction, which are generally better performing but may not perform well when there are unanticipated events not specifically considered in the modeling. We have a real-time system to evaluate the performance of the models. Therefore, we can combine the benefits of different models to achieve the overall best performance by using the model metrics in Gallery to make decisions. With a rule engine, we can define the model candidates to consider and the selection criteria for choosing a champion. At serving time, users will query Gallery for the champion model to serve based on the user-defined rules.

Model Monitoring and Retraining. After we train models and deploy model instances into production, we need to keep monitoring the performance and alert in cases like model drift, as discussed in the previous section. At the same time, model performance can degrade because the training data we used to fit the model no longer best represents the production data. Therefore, we can re-train the model with the latest training data. With a rule engine, we can set rules to automatically detect model drift, send out alerts, and trigger model training.

Model Deprecation. Models are not used forever, we may not always improve the performance of a model by retraining, and we keep experimenting with new models. When a model consistently performs worse than other models, we should deprecate it to save computational resources. Users can utilize the rule engine to define the deprecation criteria based on sustained underperformance, and training and serving costs, e.g., training takes too long or requires a large amount of computational resources. This precaution allows users to ensure that their production systems are not being negatively impacted by poorly performing models. When a model or model instance is deprecated, we would not delete them from the system, but rather flag them as deprecated. With the flag, we can skip them during model fetching or searching. Any application depending on these deprecated models or model instances can still use them until the application finishes migration to new models.

3.7.1 Rule Engine Design. To satisfy the needs of orchestrating models across lifecycle stages in production, we identify three requirements to design the rule engine: rules being *easy to understand*, *reliable*, and *agnostic to supporting services*.

Making rules easy to understand and safe to update is the first principle. We use the rules to control the production systems for model deployment and serving. We need to make sure Gallery users understand the rules and have confidence updating the rules to avoid outages due to unexpected rule usage or changes. At the same time, we need to make sure the system reliably applies the rules within a reasonable response time (SLA) when the rule is triggered. The rule engine needs also to be framework-agnostic so that any model training, monitoring, or serving components can leverage and integrate with the Gallery.

Based on frequent use cases, Gallery leverages two type of rules: *model selection rules* and *action rules*. Applying a model selection rule will return a model based on some selection criteria, e.g., returning the model that maximize AUC (area under curve). Applying an action rule will trigger some event, e.g., deploying a model instance. To make the rule engine agnostic to different frameworks within the machine learning workflow, we expect users to define callback functions that will be triggered by the rule engine. For example, to deploy a real-time forecasting model, we have one action that automatically makes a configuration change, via http request, that updates the version of the model served to consumers. There are also a default set of common actions that users can leverage or extend, like sending an email or alerting.

We use the classical *Given/When/Then* model to define rules. More concretely, for "Then" we define two templates: model selection and callback action. For example, the following rule is designed for the selection of a forecasting model.

Listing 1: Model Selection Rule Example

```
{
  "team": "forecasting",
  "uuid": "316b3ab4-2509-4ea7-8025-ca879dac61",
  "rule": {
      "GIVEN": modelName ==
        "linear_regression"
        AND model_domain == "UberX",
      "WHEN":  "metrics["mae"] <= 5",
      "ENVIRONMENT": "production",
      "MODEL_SELECTION":
      "a.created_time > b.created_time"
    }
}
```

With this rule, we select the latest trained linear regression model if the model performance is good, i.e., mae (mean absolute error) is less than 5. This rule allows the user to automatically fetch the freshest models and have confidence that the returned models are within their accepted error threshold.

The following is an example of an action rule which specifies: when we have a new model instance, and if the model performance is within a threshold, e.g., forecasting bias less than 0.1 and greater than -0.1, we can deploy the model in production.

Listing 2: Action Rule Example

```
{
  "team": "forecasting",
  "uuid": "43057544-92b0-4421-a1b0-d7d87f77a",
  "rule": {
      "GIVEN": "model_domain == "UberX"
           AND model_name == "Random Forest",
      "WHEN": "metrics.bias <= 0.1
           AND metrics.bias > -0.1",
      "CALLBACK_ACTIONS": [
        {
            "action":"forecasting_deployment"
        }
       ],
   "ENVIRONMENT": "production"
   }
}
```

3.7.2 Rule Engine Implementation. We show the overall architecture of the Gallery rule engine in Figure 8. For rule storage, we use a Git repository. To add or update a rule, users need to check it into Git within their allocated directory. The advantage of using Git is that we automatically have version control for the rules, which is critical for a production environment and enables us to set up a test framework to validate each rule before it can impact production. With Git, we can also easily enforce the peer review process for the rules to avoid production outages due to accidental updates of rules. We use Java Expression Language (JEXL) [2] to implement the rules.

The rule evaluation implementation is event based and we currently have two kinds of events to trigger the evaluation of a

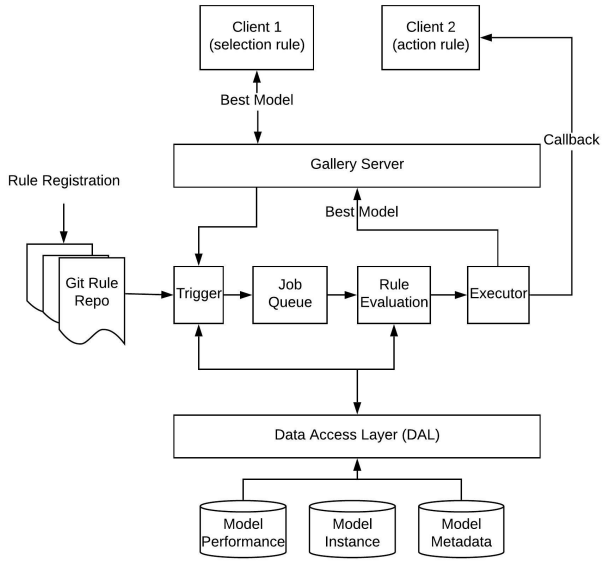

Figure 8: Gallery Rule Engine

rule: 1) directly sending a request to the rule trigger or 2) updating any metadata or metrics specific in a registered rule. Since we focus on two types of customized rules, we implement the rule evaluation ourselves instead of using other rule engines. In Figure 8, we demonstrate the workflow of the rules with two application clients. Client 1 has a model selection rule and sends the rule directly to the rule engine trigger via Gallery service. The rule is first placed in the job queue and goes through the evaluation during which the performance metrics of the related models are queried from the storage. Then the best model instance is fetched and returned to Client 1 via a Gallery service based on the condition in the rule.

Client 2 has an action rule registered in the Git rule repo. Whenever there is an update of the corresponding metadata or metric specified in the rule via a Gallery service, the evaluation of the rule will start and the job is put into evaluation job queue. If there are any action triggers, then the callback specified in the rule is executed, e.g., "when a model instance performance metric mae is less than 0.2, we deploy the model in production."

4 GALLERY IMPLEMENTATION AND USAGE CASE STUDIES

Gallery was built by the Uber Marketplace team for managing forecasting models to improve the Uber plaftform. We leverage the Uber storage infrastructure for saving the model related information and we built Gallery as a stateless microservice that includes versioning, dependency management and rule engine in Java. Gallery was designed and built to be horizontally scalable across different data centers.

Before Gallery, there was a lot of manual cost and overhead to manage our forecasting models. For about 100 models, engineers and data scientists spent 1-2 hours a day manipulating files on HDFS and Git, measuring performance and triggering model retraining. Now, Gallery has been in production for two years and is supported as part of Uber's Michelangelo ML platform. Under Michelangelo, Gallery is managing more than 1 million model instances for many machine learning applications, and engineers or data scientists no longer spend time managing files and training scripts, but instead are able to spend their energy on model iteration and experiments.

4.1 Gallery Interface

Gallery users interact with Gallery via a standard set of Thrift APIs with language-specific clients. By using Thrift, users can access the functionality of Gallery in their own modeling environment and language of their choice (e.g., Jupyter notebook, Spark application, services build in Java). With the APIs, Gallery users can manage their models cross the stages of a model lifecycle. In the following example, we use a Python application example to show one typical Gallery user workflow using the basic Gallery APIs.

```
'''
Train a ML model using SparkML and upload the
    model blob to Gallery with model instance
    metadata information.
'''
# Using a SparlML pipeline to fit a model
model_object = pipeline.fit(train_df)
# Model is serialized to a binary blob
model_content = serialize(model_object)

# Create and upload the trained model instance
    with metadata to Gallery.
model = createGalleryModel(project='example-
    project', base_version_id='supply_rejection'
    )

# Add model instance information
modelInstance = createModelInstance(model)
modelInstance.content = model_content
modelInstance.modelName = 'Random Forest'
modelInstance.city = 'New York City'
modelInstance.modelType = 'SparkML'
modelInstance.trainingDataSet = '...'
modelInstance.trainingDataMetadata = '...'
...

# Update model instance to Gallery
modelInstanceId = uploadModel(project='example-
    project', base_version_id='supply_rejection'
    , modelInstanceRecord=modelInstance
  )
```

Listing 3: Create and Save Gallery Model

The sample code in Listing 3 shows that we use SparkML to train a machine learning model and serialize the model into a binary blob. Then we add the related model instance metadata information and save the model instance to Gallery.

```
# Upload a model instance performance metric
modelPerformanceRecord = ModelEvaluationMetric(
  metricName='bias',
  scope='Validation',
  value=0.05
  )
insertModelInstanceMetric(project='example-
    project', modelInstanceId=modelInstanceId,
    modelPerformanceRecord=
    modelPerformanceRecord
  )
```

Listing 4: Save Model Performance Metric

With the model instance we have trained previously, we keep track of the model evaluation performance by saving the performance metrics in Gallery as shown in the sample code in Listing 4. At the same time, if we have one rule similar to what is shown in Listing 2 registered in the rule Git repository, then the corresponding model deployment process might be triggered based on a rule condition. How to automatically deploy model in production is different for different serving systems and we leave this part to Gallery users to define their own callback functions. For example, a realtime forecasting model is deployed in Uber by updating some configuration which is continuously monitored by the forecasting serving system in production.

```
# Model query with a given performance criteria
searchConstraint = [
    {'field':'projectId',   'operator':equal,
         'value':'example-project'},
    {'field':'modelName',   'operator':equal,
         'value':'random_forest'},
    {'field':'metricName',  'operator':equal,
         'value':'bias'},
    {'field':'metricValue', 'operator':
     smaller_than, 'value':0.25}
]
modelInstance = modelQuery(searchConstraint)
```

Listing 5: Model Search

We save the related metadata and performance metrics of the models and model instances in Gallery. Then, we could fetch the models we want with the conditions as shown in Listing 5.

4.2 Case 1: Marketplace Forecasting

At Uber, the Marketplace Forecasting team generates real-time and long-term forecasts for multiple applications, including driver suggestions and pricing [4]. Multiple supply and demand models have evolved through different model classes ranging from simple time series models, linear regression models, and now deep learning models. Each model class is trained and deployed per city Uber operates in. Each city faces different market dynamics, and classes of models perform differently based on certain spatial or temporal characteristics of the city. Therefore, the team needs a mechanism to track and train each model's performance on a per city basis, and a systematic way to determine which model class to serve at a given time. As a result, the Marketplace Forecasting team alone has thousands of model instances to maintain and decisions to make each minute about which model to serve. Gallery's model management solution with storage and automation via rule engine has reduced model deployment from two hours of engineering work per model to 0.

Another use case is dynamic model switching for forecasts when there are events e.g., holidays. Via action rules, Gallery is able to inform forecasting serving system about the performance of models that include holiday/event features versus those that do not, and subsequently switch to serve the appropriate models for the duration of the event. This improves the accuracy of the served predictions by more than 10% MAPE (Mean Absolute Percent Error) compared to a static served model. Furthermore, Model Health alerts continue to monitor the performance of such models and can alert engineers regarding issues with prediction accuracy. These alerts have proven useful in the case of unplanned events (e.g., public transit outages) that cause unexpected spikes in demand, and gives engineers or ops an opportunity to intervene and mitigate the performance degradation.

4.3 Case 2: Marketplace Simulation Platform

The Marketplace Simulation platform [5] hosts a simulated world with driver-partners and riders, mimicking scenarios in the real world. Leveraging an agent-based discrete event simulator, the platform allows Uber Marketplace engineers and data scientists to rapidly prototype and test new features and hypotheses in a risk-free environment.

Priori to leveraging Gallery, one issue the simulation platform had is model reusability. ML developers implemented models directly in the simulator and trained them on the fly as the simulator ran. As a result, the complexity of the system and the wide array of models being simulated degraded the performance of the platform.

Gallery became part of the solution by providing the simulation platform with metadata and model binary storage, which enabled the platform to decouple model training and serving. Offline processes can store reusable model instances into Gallery, and the simulation backend service can instantiate such models as they're needed. This decoupling allows model developers to iterate and evolve their models, independently of the simulator's backend, whereas before they need to wait for the entire end-to-end process each time they trained/updated a model. Once a model developer is satisfied with their model, they can store their model in Gallery, which will signal to the backend service the presence of a new model. Furthermore, decoupling model training workloads from the simulator, allowed the Simulation team to simplify the complexity of the simulator reduce maintenance costs, conserve hardware resources while improving system efficiency. The Gallery system has saved the simulation platform an estimated 8GB memory and one hour CPU time per simulation.

5 RELATED WORK

With the proliferation of Big Data and large-scale computing (e.g., MapReduce [16], Apache Spark[31]), several scalable machine learning platforms [32] have emerged in recent years with the focus of machine learning training on a large amount of data. Apache Spark is a general data processing framework. MLlib [20, 23], the machine learning library added to Apache Spark, makes Apache Spark broadly used for some simple large-scale machine learning model training. However, for complex machine learning tasks, especially deep learning [17], which requires state updates and iteration, parameter server architecture is used for enabling in-place updates for very large parameters, e.g., Parameter Server [18], PMLS [30], Google DistBelief [15]. TensorFlow [11] and MXNet [24] are advanced machine learning frameworks focusing more on the deep learning problem and can fully utilize different devices like CPU and GPU.

Managing a large number of machine learning models in production is challenging. There is an ever-increasing interest in the problem and several academic and industry efforts have been published. ModelDB [28] is an open source model management system which provides APIs for saving models and associated metadata, measuring performance, and querying models. It has a web-frontend for easy visualization and summary of the model information. ModelDB also provides clients with the ability to integrate its model management features with Apache Spark ML and scikit-learn [25]. However, there is no orchestration of model training, serving and deployment in ModelDB.

A lightweight system is proposed in [27] to extract, store, and manage machine learning model metadata. It tracks the provenance information of datasets, models, predictions, evaluations

Table 1: Model Management System Comparison (Y: Yes, N: No)

Systems	Saving	Loading	Metadata	Searching	Serving	Metrics	Orchestration
ModelDB [28]	Y	Y	Y	Y	N	Y	N
ModelHUB [21]	Y	Y	Y	Y	N	Y	N
Metadata Tracking [27]	N	N	Y	Y	N	Y	N
Velox [13]	Y	Y	Y	N	Y	Y	Y
Clipper [14]	Y	Y	N	N	Y	Y	Y
MLFlow [22]	Y	Y	Y	Y	Y	Y	N
TFX [12]	Y	Y	Y	N	Y	Y	Y
Azure ML [1]	Y	Y	N	N	Y	N	Y
SageMaker [26]	Y	Y	N	N	Y	N	Y
Gallery	Y	Y	Y	Y	N	Y	Y

and training runs in machine learning experiments. With model metadata and provenance, the system provides a basis for comparability and repeatability of machine learning experiments. Similar to ModelDB, there is no orchestration of machine learning stages in this system. This system is focusing on the metadata associated with model experiments instead of managing the machine learning models.

ModelHUB [21] was built to manage the lifecycle of deep learning models. Considering the huge space of potential deep learning models by tweaking neural network architectures and hyperparameters, ModelHUB compactly stores a large number of models and snapshots with fast query performance. It also keeps track of the model metadata including the model accuracy score. ModelHUB focuses on deep learning models and is not framework agnostic.

Velox [13] is a low-latency and large-scale model serving system. It focuses on making the model serving efficient by caching computation and scaling out model prediction. Velox leverages a cluster computation framework and incremental model updates to scale out the model training process. Velox also manages the model lifecycle by detecting model performance degradation to trigger model rollback or re-training. However, the project was deprecated [29].

Clipper [14], the follow-up project of Velox, is a general-purpose low-latency prediction serving system. It can incorporate multiple machine learning frameworks including TensorFlow [11], Apache Spark [31] and scikit-learn [25]. Similar to Velox, it uses caching, as well as batch and adaptive model selection to improve model prediction latency and performance. Clipper focuses on serving models with low latency across different frameworks.

MLflow [22] is an open source platform under active development for managing the machine learning lifecycle. There are three major components in MLflow: MLflow Tracking, which tracks the experiments results and parameters; MLflow Project, which packages the ML code to be easily reproducible; and MLFlow Model, which provides a standard format for packaging ML models across different libraries or framework. The same model could be packed with different flavors such that a model could be applied in different frameworks, e.g., a TensorFlow model can be used with TensorFlow flavor or python function flavor. With this MLflow Model format, the models can be used in a variety of downstream tools, e.g., real-time serving through a REST API or batch inference on Apache Spark. MLFlow also provides CLI to run the MLflow models for model deployment. However, there is no orchestration to coordinate the moving of models across different stages in a model lifecycle.

TFX [12] is a production-scale machine learning platform for TensorFlow and it consists of multiple components for machine learning, including data transformation, model training, model evaluation, and model serving. With the TensorFlow serving component [9], TensorFlow models can be deployed and served in production. However, TensorFlow serving is a not generic component for managing a variety of machine learning models using different frameworks. Kubeflow [3] is a project to make deploying ML workflows on Kubernetes simple, portable, and scalable. It started as an open source project from Google that highlighted how the company ran Tensorflow internally based on TFX. Now, Kubeflow has extended to be a multi-architecture, multi-cloud framework for running entire ML pipelines.

Azure ML [1] is a machine learning platform where we can process data, build models and publish and stage a predictive model as an Azure-based service. Similar to Azure ML [1], AWS SageMaker [26] also provides the functionality to build models, train models, and deploy models in production. Both Azure ML and AWS SageMaker are closed systems making single component integrations challenging. The model management in Azure ML and AWS SageMaker is really focused on training a model and deploying a model in its own system. They are not model neutral and framework agnostic. Kepler [19] and Taverna [10] are popular scientific workflow systems which manage complex data analytics pipelines including data access, data analysis and mining steps, and many other steps including computationally intensive jobs on high-performance cluster computers.

We present a feature comparison of different model management systems in Table 1.

6 LESSONS LEARNT IN BUILDING GALLERY

6.1 Common ML Tools

Machine learning is becoming the essential component of many Uber product features. Accordingly, more and more teams at Uber are using machine learning or beginning to use machine learning. Different teams might be at different maturity stages of applying machine learning depending on the team's experience, but all of them will go through solving the common issues of managing the models in a machine learning application. Without shared common ML infrastructure tools, each team might waste a lot of resources to "reinvent the wheel." When we built Gallery, many teams express the similar needs of Gallery to manage their models. As a result, we made Gallery part of Uber's standard ML infrastructure as part of Michelangelo so that it could benefit all the teams at Uber. It also shows the importance of building common ML tools so that all the product teams can boost their

productivity by focusing on their own business problems and model iterations without worrying about how to train, manage and serve their models.

6.2 Model Reproducibility

Shortly after onboarding the first group of users of Gallery, we observed and learned that one of the more desired features was model reproducibility. Reproducibility was not one of the original use cases we had in mind when we built Gallery. Instead, we had focused directly on performance tracking and alerting. However, it became clear that a user's natural follow-up to an alert is to attempt to reproduce the problem, and it was apparent a model management system needs to support this functionality. The original Gallery data model did not include many of the metadata components included today that led to model reproducibility (e.g., training data information, training frameworks and features), but an important lesson learned is that reproducibility is central to supporting the ML model lifecycle. Users need the ability to recreate models or replay history in order to understand their production flows and debug performance. Gallery has proven to be a valuable tool for model builders at Uber in simplifying the model debugging process.

6.3 Tiered Service Offering

Another lesson we have learned during the development of Gallery is how to make user adoption of such tools easy. As discussed before, there is a wide variety of ML tooling being used at Uber and teams across the company are working against their own deadlines with different features in their tech stacks. Therefore, there would be no single opportunity to ask users to migrate their workflows and no central mechanism by which we could directly onboard them to Gallery. Instead, we opted to provide a tiered set of features and solutions that teams could leverage as they had the bandwidth and discovered the need. We wanted our features to be modular in that users at any point in their maturity could leverage the system, with the opportunity to add more complex functionalities in the future.

Gallery features are broken up into four groups that are built on top of one another: 1) model storage and retrieval; 2) metadata storage and search; 3) metric storage and search; and 4) rule engine automation. As teams start to use Gallery in their systems, they sequence their onboarding based on the features and complexity that they need to unlock. For teams exploring new modeling techniques or building a system from scratch, it is often the case that they only need feature set 1), and optionally 2). Teams doing experimentation have not yet thought about automation and only need a place to dump models to rapidly do more testing. However, once teams have built out a model and are trying to scale to meet business requirements, they then see the need for feature sets 3) and 4). By using the base functionality of data storage and retrieval in their experimentation, there is only an incremental additional effort required to unlock more complex Gallery features that help to automate entire workflows. This approach helped Gallery gain quick adoption among five teams with in its first six months.

7 CONCLUSIONS

In this paper, we describe the machine learning model management problem across the different stages of a model's lifecycle for a large number of models and model instances in production environments at Uber. We describe the model lifecycle management system, Gallery, a solution used in production to manage machine learning models across different services at Uber.

Design for system automation up front is critical to manage thousands of machine learning models in production. Developing and applying machine learning models involves multiple stages across a model's lifecycle. Manually managing the models and model instances in production is not scalable and is error-prone. Building generic systems to be able collect and keep track of model and instance information, as well as dependencies is critical for maintaining accurate production systems. On top of the raw information, we can produce intelligence. With the help of rules, we can orchestrate the whole modeling workflow, which dramatically boosts data scientists and engineers' productivity and also makes the machine learning systems more reliable and scalable.

Building an agnostic model management system is critical for adoption and user on-boarding. At Uber, there are a large number of existing machine learning applications, which often leverage different languages and frameworks for model development and serving. Building Gallery to be agnostic to machine learning frameworks has allowed the system to be adopted by many teams at Uber and has helped the company to align on a common infrastructure for the machine learning model management.

8 ACKNOWLEDGMENTS

We would like to sincerely recognize and acknowledge the following contributors who played a significant role in the design, development, and productionization of Gallery at Uber: Anne Holler, Yifan Ma, Luke Duncan, Mani Bhushan, Aparna Dhinakaran, and Zhenghui Wang. Additional contributors to the project that we would like to thank also include Andrii Iasynetskyi, Rashmi Venugopal, and Steven Wright. We would also like to thank Haoyang Chen, from the Simulation Platform, for being the first to adopt Gallery and providing extensive feedback to improve the design of the system.

REFERENCES

[1] Azure ML. https://studio.azureml.net.
[2] JEXL. https://commons.apache.org/proper/commons-jexl/.
[3] Kubeflow. https://www.kubeflow.org.
[4] Machine learning at uber. https://eng.uber.com/machine-learning.
[5] Marketplace Simulation. https://eng.uber.com/simulated-marketplace/.
[6] Michelangelo. https://eng.uber.com/michelangelo/.
[7] Rules of machine learning. https://developers.google.com/machine-learning/guides/rules-of-ml/.
[8] Semantic Versioning. https://semver.org/.
[9] TensorFlow Serving. https://www.tensorflow.org/serving.
[10] The taverna workflow suite: designing and executing workflows of web services on the desktop, web or in the cloud. *Nucleic Acids Res*, 41, 2013.
[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. OSDI'16.
[12] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. TFX: A TensorFlow-based production-scale machine learning platform. KDD '17.
[13] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. CIDR'15.
[14] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. NSDI'17.
[15] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. NIPS'12.
[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 2008.

[17] Y. LeCun, Y. Bengio, and G. E. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
[18] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. OSDI'14.
[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10), 2006.
[20] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine learning in apache spark. *CoRR*, 2015.
[21] H. Miao, A. Li, L. S. Davis, and A. Deshpande. ModelHub: Deep learning lifecycle management. ICDE'17.
[22] MLFlow. https://mlflow.org.
[23] MLlib. http://spark.apache.org/mllib.
[24] MXNet. https://mxnet.apache.org.
[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 2011.
[26] SageMaker. https://aws.amazon.com/sagemaker.
[27] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. In *NIPS Workshop ML Systems*, 2017.
[28] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. ModelDB: A system for machine learning model management. HILDA '16, 2016.
[29] Velox. https://github.com/amplab/velox-modelserver.
[30] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. KDD '15.
[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. HotCloud'10.
[32] K. Zhang, S. Alqahtani, and M. Demirbas. A comparison of distributed machine learning platforms. ICCCN '17.