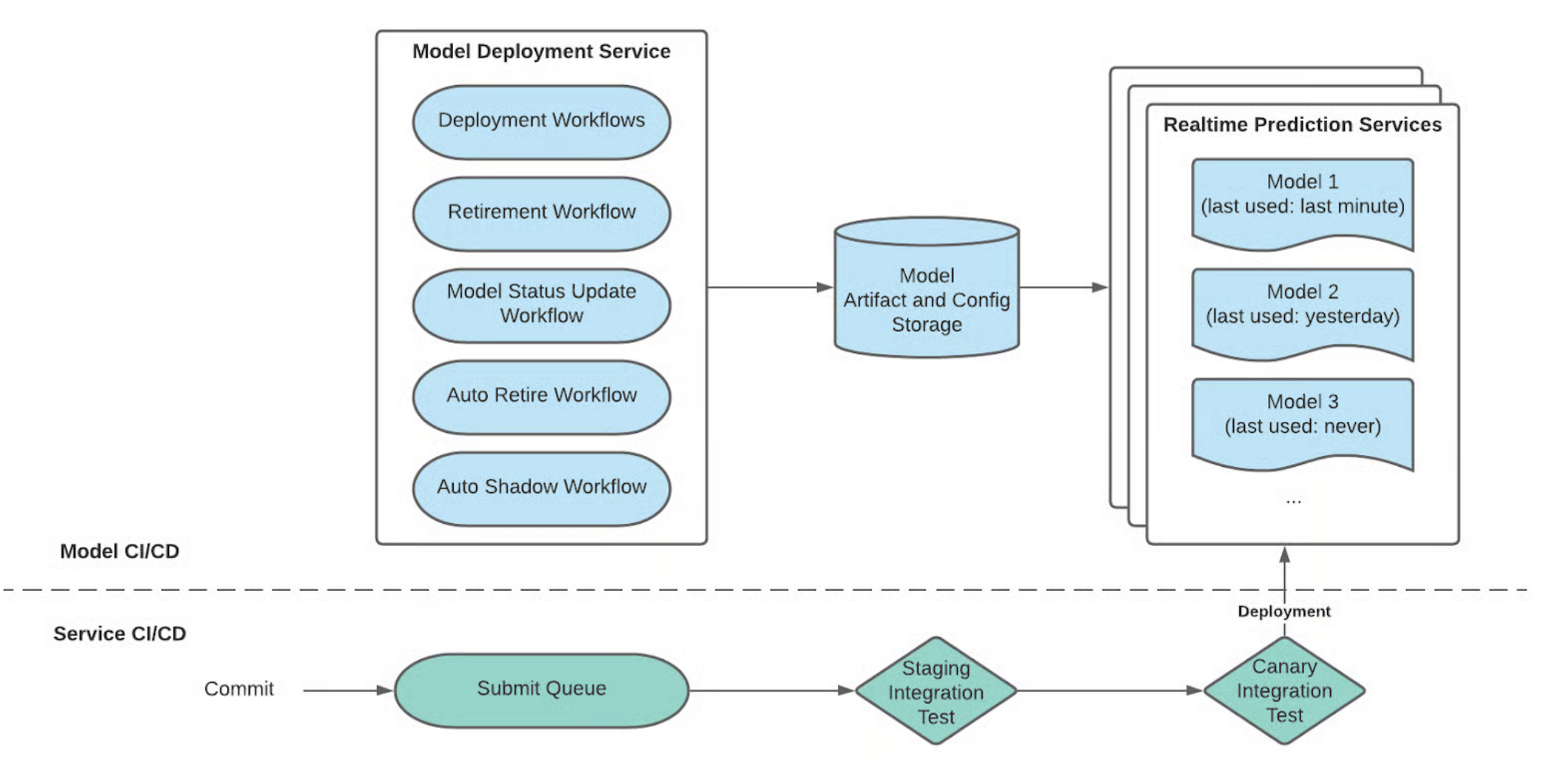
Data / ML, Engineering

Continuous Integration and Deployment for Machine Learning Online Serving and Models

30 June 2021 / Global



Introduction

At Uber, we have witnessed a significant increase in machine learning adoption across various organizations and use-cases over the last few years. Our machine learning models are empowering a better customer experience, helping prevent safety incidents, and ensuring market efficiency, all in real time. The figure above is a high level view of CI/CD for models and service binary.

One thing to note is we have continuous integration (CI)/continuous deployment (CD) for models and services, as shown above in Figure 1. We arrived at this solution after several iterations to address some of MLOps challenges, as the number of models trained and deployed grew rapidly. The first challenge was to support a large volume of model deployments on a daily basis, while keeping the Real-time Prediction Service highly available. We will discuss our solution in the Model Deployment section.

The memory footprint associated with a Real-time Prediction Service instance grows as newly retrained models get deployed, which presented our second challenge. A large number of models also increases the amount of time required for model downloading and loading during instance (re)start. We observed a great portion of older models received no traffic as newer models were deployed. We will discuss our solution in the Model Auto-Retirement section.

The third challenge is associated with model rollout strategies. Machine learning engineers may choose to roll out models through different stages, such as shadow, testing, or experimentation. We observed some common patterns in model rollout strategies and decided to incorporate them into the Real-time Prediction Service, which we will discuss in the Auto-Shadow section.

As we are managing a fleet of Real-time Prediction Services, manual service software deployment is not an option. The fourth challenge is to have a CI/CD story for Real-time Prediction Services software. During model deployment time, the Model Deployment Service performs validation by making prediction calls to the candidate model with sampled data. However, it does not check against existing models deployed to the Real-time Prediction Services. Even if a model passes validation, there is no guarantee that the model can be used or exhibits the same behavior (for feature transformation and model evaluation) when deployed to production Real-time Prediction Service instances. This is because there could be dependency changes, service build script changes, or interface changes between two Real-time Prediction Service releases. We will discuss our solution in the Continuous Integration and Deployment section.

Model Deployment

For managing the models running in the Real-time Prediction Service, machine learning engineers can deploy new and retire unused models through the model deployment API. They can track their models' deployment progress and health status through the API. You can see the system's internal architecture below in Figure 2:

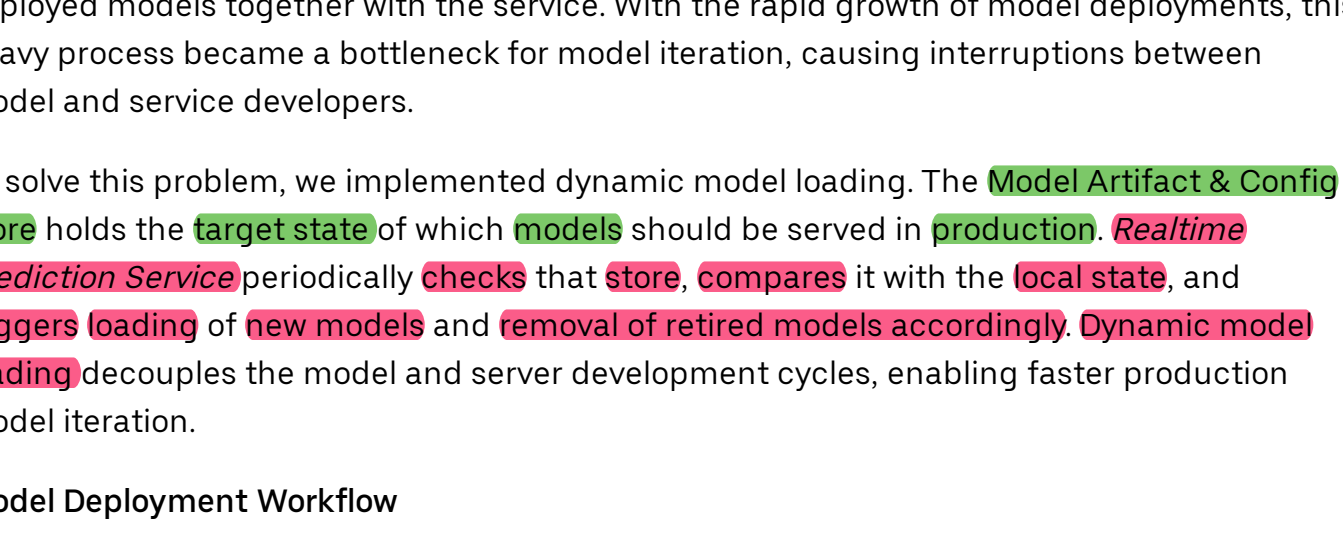


Figure 2: Model Deployment Workflow and Health Check Workflow

Dynamic Model Loading

Historically, we sealed model artifacts into Real-time Prediction Service docker images, and deployed models together with the service. With the rapid growth of model deployments, this heavy process became a bottleneck for model iteration, causing interruptions between model and service developers.

To solve this problem, we implemented dynamic model loading. The Model Artifact & Config store holds the target state of which models should be served in production. *Realtime Prediction Service* periodically checks that store, compares it with the local state, and triggers loading of new models and removal of retired models accordingly. Dynamic model loading decouples the model and server development cycles, enabling faster production model iteration.

Model Deployment Workflow

Model deployment does not simply push the trained model into Model Artifact & Config store; it goes through the steps to create a self-contained and validated model package:

- Artifacts validation: makes sure the trained model includes all necessary artifacts for serving and monitoring
- Compile: packages all the model artifacts and metadata into a self-contained and loadable package into the Real-time Prediction Service
- Serving validation: load the compiled model jar locally and perform a model prediction with example data from training dataset—this step ensures that the model can run and is compatible with Real-time Prediction Service

The primary reason for these steps is to ensure the stability of the Real-time Prediction Service. Since multiple models are loaded in the same container, a bad model may cause prediction request failure, and potentially interrupt models on the same container.

Model Deployment Tracking

For helping machine learning engineers manage their production models, we provide tracking for deployed models, as shown above in Figure 2. It involves two parts:

1. **Deployment progress tracking:** Deployment workflow will post deployment progress updates to the centralized metadata storage for tracking.
2. **Health check:** After the model finishes its deployment workflow, it becomes a candidate for model health check. The check is performed periodically to track model health and usage information, and it sends updates to the metadata storage.

Model Auto-Retirement

There is an API for retiring unused models. However, in many cases people forget to do that, or do not integrate model cleanup into their machine learning workflows. This results in unnecessary storage costs and an increased memory footprint. A large memory footprint can cause Java garbage collection pauses and out-of-memory errors, both of which can impact quality of service. To address this, we built a model auto-retirement process, wherein owners can set an expiration period for the models. If a model has not been used beyond the expiration period, the Auto-Retirement workflow, in Figure 1 above, will trigger a warning notification to the relevant users and retire the model. We saw a non-trivial reduction in our resource footprint after we enabled this feature.

Auto-Shadow

As machine learning engineers choose to roll out models with different strategies, they often need to devise ways to distribute real-time prediction traffic among a set of models. We have seen common patterns, such as gradual rollout and shadowing, in their traffic distribution strategies. In a gradual rollout, clients fork traffic and gradually shift the traffic distribution among a group of models. In shadowing, clients duplicate traffic on an initial (primary) model to apply on another (shadow) model). Figure 3 illustrates a typical traffic distribution among a set of models, wherein models A, B, and C participate in a gradual rollout, while model D shadows model B.

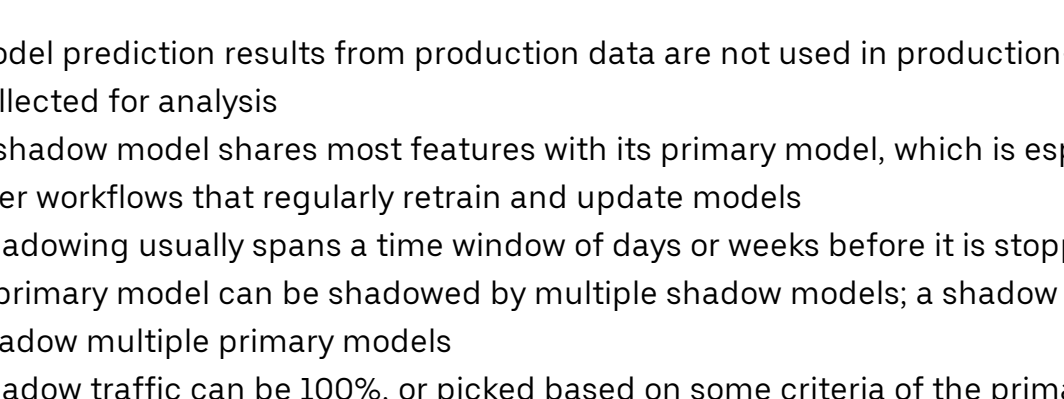


Figure 3: Real-time prediction traffic distribution among a set of models

To help reduce engineering hours on developing repetitive implementations for common patterns, Real-time Prediction Services provides built-in mechanisms for traffic distribution. Below we focus on the scenario of model auto-shadowing.

Different teams have different strategies for model shadowing, but they all share common characteristics:

- Model prediction results from production data are not used in production—they are collected for analysis
- A shadow model shares most features with its primary model, which is especially true in user workflows that regularly retrain and update models
- Shadowing usually spans a time window of days or weeks before it is stopped
- A primary model can be shadowed by multiple shadow models; a shadow model can shadow multiple primary models
- Shadow traffic can be 100%, or picked based on some criteria of the primary model traffic
- To compare the results, the same prediction is collected for both the primary and the shadow models.
- A primary model may be serving millions of predictions, and prediction logs may be sampled

We add auto-shadow configuration as part of the model deployment configurations. Real-time Prediction Service can check on the auto-shadow configurations, and distribute traffic accordingly. Users only need to configure shadow relations and shadow criteria (what to shadow and how long to shadow) through API endpoints, and make sure to add features that are needed for the shadow model but not for the primary model.

We found a built-in auto-shadow feature provides extra benefits:

- As the majority of primary and shadow models share a set of common features, Real-time Prediction Service only fetches features from the online feature store that are not used in the primary model for the shadow models
- By combining built-in prediction logging logic and shadow sampling logic, Real-time Prediction Service can reduce the amount of shadow traffic to those that are destined to be logged
- Shadow models can be treated as second class models when service is under pressure, and paused/resumed in order to relieve load pressure

Continuous Integration and Deployment

We rely on CI/CD for service release deployment for a fleet of Real-time Prediction Services. Since we are supporting critical business use cases, in addition to validation during model deployment, we need to ensure high confidence in the automated continuous integration and deployment process.

Our solution tries to address the following issues with new release:

- **Code change not compatible**: there could be two symptoms for this issue—a model is unable to load or make predictions with new binaries, or its behavior changed with new releases. The latter is difficult to identify and fix, and is critical to a model's correctness.
- **Dependencies not compatible**: service unable to start due to underlying dependency changes.
- **Build script not compatible**: release not able to build due to build script changes.

To address the above issues, we employed a three-stage strategy for validating and deploying the latest binary of the Real-time Prediction Service: staging integration test, canary integration test, and production rollout. The staging integration test and canary integration tests are run against non-production environments. Staging integration tests are used to verify the basic functionalities. Once the staging integration tests have been passed, we run canary integration tests to ensure the serving performance across all production models. After ensuring that the behavior for production models will be unchanged, the release is deployed onto all Real-time Prediction Service production instances, in a rolling deployment fashion.

Final Thoughts

We have shared our solutions for some of MLOps challenges. As we evolve Uber's machine learning infrastructure and platform and support new machine learning use cases, we see new MLOps challenges emerge. Some of the things that we are working on include: near real-time monitoring for inference accuracy, feature quality, and business metrics; deploy and serve multi-task learning and hybrid models; perform feature validation; better model fallback mechanism; model traceability and debuggability, etc. Stay tuned.

Acknowledgements

We could not have accomplished the technical work outlined in this article without the help of our team of engineers and data scientists at Uber. Special thanks to Smitha Shyam and the entire Michelangelo team for their support and technical discussion.

Joseph Wang
Joseph Wang serves as a Principal Software Engineer on the AI Platform team at Uber, based in San Francisco. His notable achievements encompass designing the Feature Store, expanding the capacity of the real-time prediction service, developing a robust model platform, and improving the performance of key models. Presently, Wang is focusing his expertise on advancing the domain of generative AI.

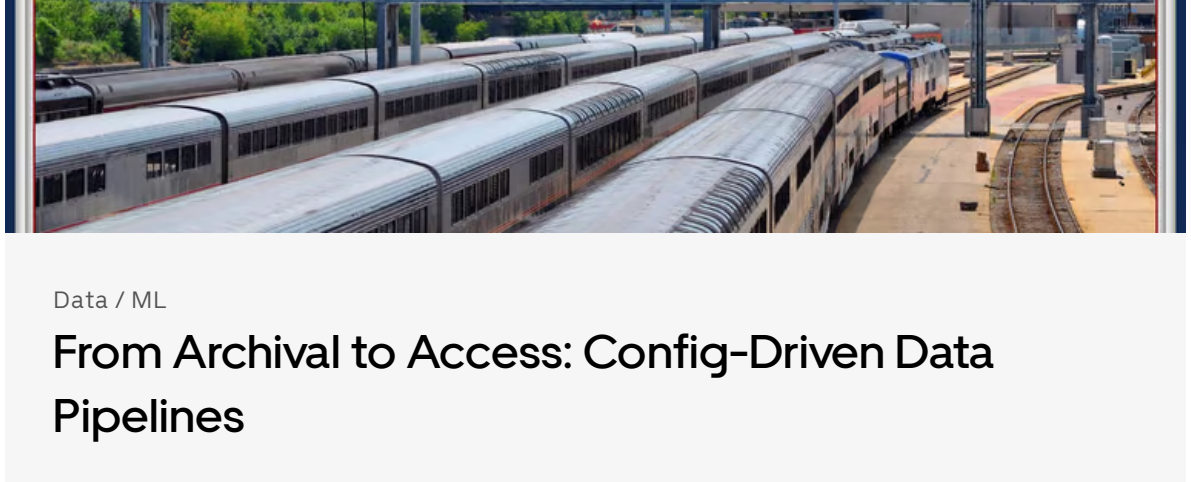
Jia Li
Jia Li is a Senior Software Engineer on Uber's Machine Learning Platform team. He works on model deployment, realtime prediction, and model monitoring.

Yi Zhang
Yi Zhang is a Senior Software Engineer on Uber's Machine Learning Platform team. She thrives while solving big data problems, from data infrastructure to data application layers.

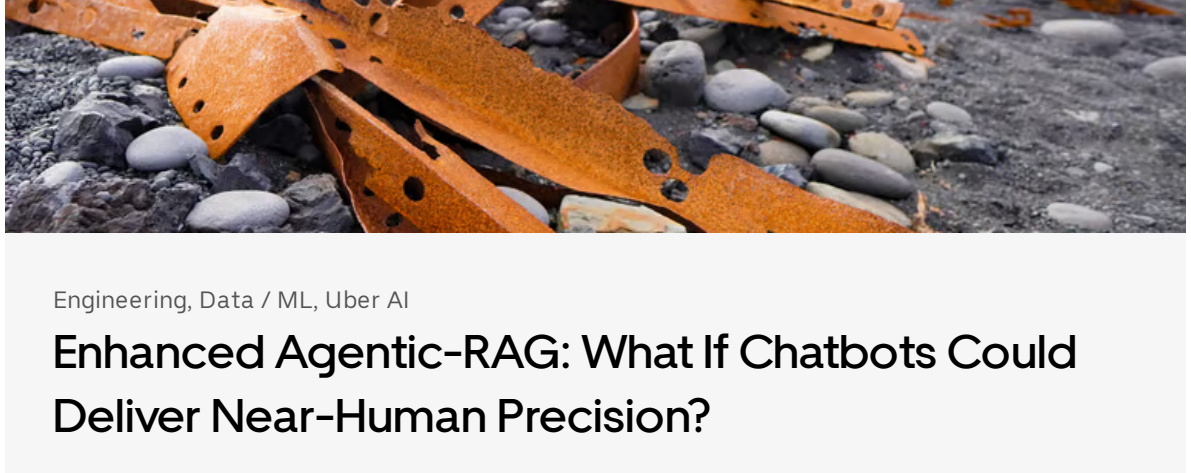
Yunfeng Bai
Yunfeng Bai was a TLM on Uber's Machine Learning Platform team. He led the team on model management and real-time prediction related efforts.

Posted by Joseph Wang, Jia Li, Yi Zhang, Yunfeng Bai
Category: Data / ML Engineering

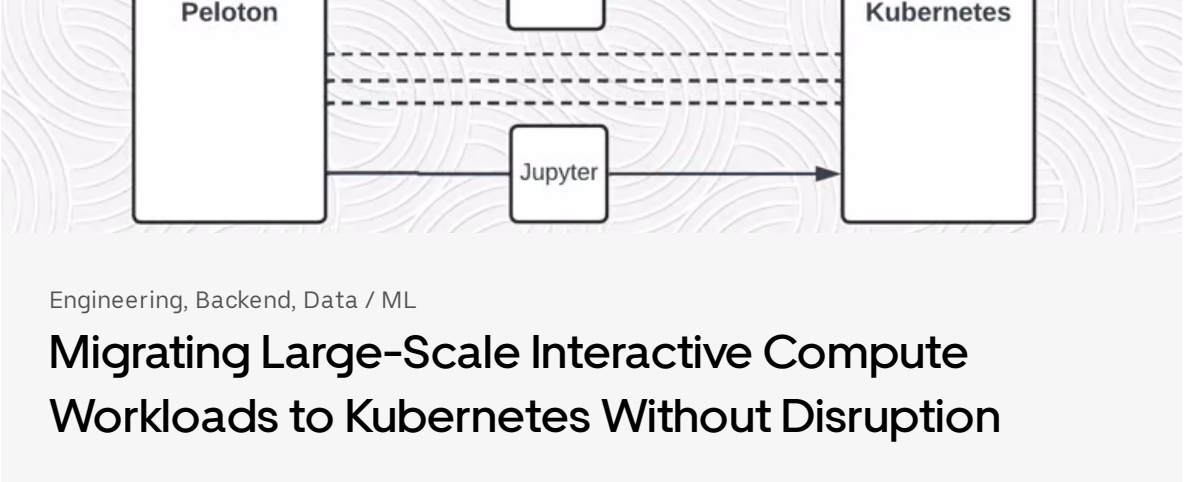
Related articles



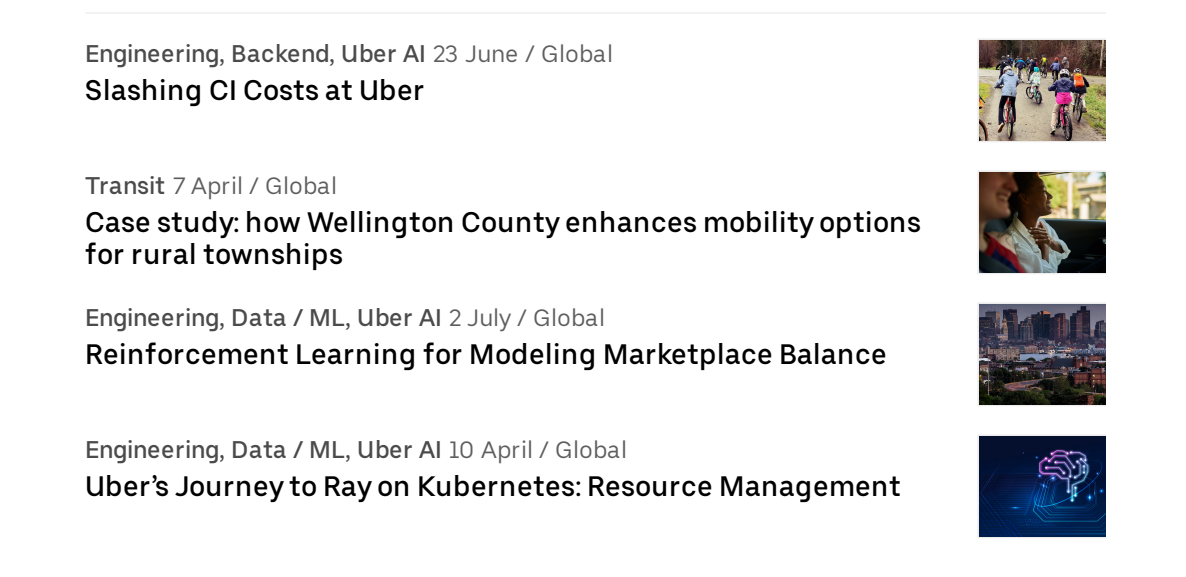
Engineering, Data / ML, Uber AI
Reinforcement Learning for Modeling Marketplace Balance
2 July / Global



Engineering, Data / ML
How Uber Migrated from Hive to Spark SQL for ETL Workloads
12 June / Global



Data / ML
From Archival to Access: Config-Driven Data Pipelines
5 June / Global



Engineering, Data / ML, Uber AI
Enhanced Agentic-RAG: What If Chatbots Could Deliver Near-Human Precision?
29 May / Global

Engineering, Backend, Data / ML
Migrating Large-Scale Interactive Compute Workloads to Kubernetes Without Disruption
8 May / Global

Most popular

Engineering, Backend, Uber AI 23 June / Global
Slashing CI Costs at Uber

Transit 7 April / Global
Case study: how Wellington County enhances mobility options for rural townships

Engineering, Data / ML, Uber AI 2 July / Global
Reinforcement Learning for Modeling Marketplace Balance

Engineering, Data / ML, Uber AI 10 April / Global
Uber's Journey to Ray on Kubernetes: Resource Management

View more stories

English

Reviews

GET IT ON Google Play

Download on the App Store

Privacy Accessibility Terms