Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Automated MLOps Retraining Pipeline for LLM and Recommendation Systems

Addressing Data and Concept Drift in Production AI

Mastan Vali
Department of Computer Science and Engineering
Sreenidhi Institute of Science and Technology
Hyderabad, India

Shaik Riyan Nabi
Department of Computer Science and Engineering
Sreenidhi Institute of Science and Technology
Hyderabad, India

Meghana Batchalakuri
Department of Computer Science and Engineering
Sreenidhi Institute of Science and Technology
Hyderabad, India

K. Navya Sai
Department of Computer Science and Engineering
Sreenidhi Institute of Science and Technology
Hyderabad, India

Abstract **- Large Language Models and recommendation systems have a problem. They do not work well over time because the data they are using is changing. This is called data drift and concept drift. Studies have shown that Large Language Models can be 15-30% less accurate after 6-12 months. Recommendation systems can be 10-25% less effective after 3-6 months. This paper is a way to automatically update these systems so they keep working well. It uses something called an MLOps retraining pipeline. This pipeline checks the systems to see if they are working correctly. If they are not it updates them without needing a person to do it. The system has a parts. It has a web application that people can use to get movie and product recommendations. It has a database to store information. It has a part that watches how well the recommendations are working.. It has a part that updates the system if it is not working well. For Large Language Models the system uses something called LoRA tuning. This makes it faster. Uses less memory. For recommendation systems it uses something called collaborative filtering. This helps the system get better at making recommendations. The system also has a way to make sure it is working correctly before it is fully used. This is called deployment. It starts by using the system with just a few people and then slowly adds more. If there is a problem it can go back, to the system. The results of this system are very good. It makes recommendations that're 34% better. People click on the recommendations 28% often.. It can update the system automatically 92% of the time. This saves a lot of work because it does not need people to update it. It is a way to keep these systems working well and it can be used in many places. Large Language Models and recommendation systems can be very helpful if they are working correctly.**

Keywords – MLOps, Data Drift, Concept Drift, LLM Fine-Tuning, Recommendation Systems, Auto-Retraining; LoRA, Collaborative Filtering, Canary Deployment, Model Monitoring

I. INTRODUCTION

A. Background and Motivation

The use of machine learning models in world situations creates a big problem. The information that these models learn from is not the same as the information they get when they are being used. This happens because the models do not change. The information they get does. There are two things that can happen because of this. One is called data drift. This is when the information that the model gets changes over time. For example the way people use things can. The words they use can be different. The other thing is called concept drift. This is when what the model thinks is important is not the same as what's really important.

What people like today is not the same as what they liked six months If we do not do anything about this problem it can cause issues. Big language models can get worse at their job by 15 to 30 percent in six to twelve months if they are not updated. This is because the information they learned is old and the words they know are not the same as the words people use now. Recommendation systems can also get worse by 10 to 25 percent in three to six months because peoples likes and dislikes change and new things are added all the time.

In both cases models that are not updated cannot keep giving people what they want which is important, for businesses. Usually when models get worse people have to update them by hand. Someone has to watch how the model is doing see that it is not working well schedule an update check that the update is good and then put the model in place. This takes a time and needs special people to do it. It is also not a way to do things because it is only done after the model has already gotten worse. With many people using these models updating them by hand is not a good idea.

This paper presents an automated MLOps retraining pipeline that eliminates the human-in-the-loop dependency from the model update cycle. The system continuously monitors production model performance, applies statistical drift detection, automatically triggers retraining when degradation thresholds are crossed, and deploys updated models through a safe canary rollout process. The pipeline supports both LLM fine-tuning via LoRA and recommendation system retraining via incremental collaborative filtering and two-tower neural architectures.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

B. Research Contributions

The primary contributions of this work are:

- Design and implementation of a complete automated MLOps retraining pipeline integrating drift detection, orchestrated retraining, model versioning, and canary deployment in a unified production-ready architecture.
- Application of parameter-efficient LoRA fine-tuning (rank=16) to LLM adaptation under drift, achieving 65% reduction in retraining cycle time and 50-70% reduction in GPU memory requirements relative to full fine-tuning.
- Implementation of a multi-signal drift detection framework combining KL divergence, Population Stability Index (PSI), and Kolmogorov-Smirnov testing for robust statistical detection of both data and concept drift in production recommendation streams.
- Experimental validation demonstrating 34% improvement in recommendation accuracy, 28% increase in CTR, and 92% end-to-end automation rate on a Flask-PostgreSQL-backed movie and product recommendation system.
- Safe canary deployment protocol with progressive traffic routing (5%, 25%, 50%, 75%, 100%) and automatic rollback, maintaining 99.9% production uptime throughout model transitions.

II. LITERATURE SURVEY

The challenge of model drift in production systems has been recognized in the machine learning literature since the early deployment of online learning systems. Widmer and Kubat provided one of the foundational treatments of concept drift, proposing adaptive windowing mechanisms to detect distribution changes in streaming data. Their work established the theoretical basis for threshold-based drift detection that underlies most contemporary production monitoring systems, including the automated retraining pipeline presented here. [1]

Brown et al. demonstrated that large-scale language models pre-trained on broad corpora exhibit strong few-shot generalization capabilities, establishing that model scale and pre-training data diversity are critical determinants of downstream task performance. Their findings directly motivate the need for domain-adaptive retraining strategies in production LLM deployments, as pre-trained distributions progressively diverge from evolving real-world interaction data over time. [2]

Sculley et al. characterized the hidden technical debt that accumulates in machine learning systems deployed in production environments, identifying data dependency, feedback loops, and undeclared consumers as primary sources of system fragility. Their analysis argued that continuous monitoring and disciplined pipeline engineering are necessary to prevent silent model degradation. These are concerns that directly motivate the automated drift detection, retraining orchestration, and rollback mechanisms designed here. [3]

Bifet and Gavalda introduced the ADWIN (Adaptive Windowing) algorithm, which provides provably correct detection of distribution changes in data streams with bounded false positive rates. By maintaining a variable-length sliding window whose size adapts to the observed rate of change, ADWIN avoids the manual window-size tuning required by fixed-window detectors, and serves as a foundational statistical method underlying the drift detection component of the proposed pipeline. [4]

Baena-Garcia et al. proposed the Early Drift Detection Method (EDDM), which improves sensitivity to gradual concept drift by monitoring changes in the error-rate distance between consecutive classifier predictions. Building on such work, KL divergence and Population Stability Index (PSI) have become the dominant operational metrics in industry for detecting input feature distribution shift, largely due to their computational efficiency and interpretability by non-specialist operators managing production deployments. [5]

Hu et al. introduced LoRA (Low-Rank Adaptation), demonstrating that the weight updates required for domain-specific adaptation of large language models have an intrinsically low-rank structure. By training only two small decomposition matrices per target weight layer rather than all model parameters, LoRA achieves parity with full fine-tuning on domain adaptation benchmarks while reducing trainable parameters by over 10,000-fold — an efficiency that is critical in the automated retraining pipeline presented here, where GPU compute cost per retraining cycle is a binding operational constraint. [6]

Koren, Bell, and Volinsky established matrix factorization as a central technique for collaborative filtering in recommender systems, showing that Alternating Least Squares (ALS) decomposition of the user-item interaction matrix yields compact latent factor representations that outperform neighbourhood-based methods. Their framework supports incremental user and item embedding updates as new interactions arrive without requiring full re-decomposition, making it well-suited to the continuous retraining regime addressed in this paper. [7]

Covington, Adams, and Sargin described the two-tower deep neural network architecture deployed for recommendations at YouTube, in which separate encoder networks independently produce user and item embeddings that are compared via dot product at inference time. This architecture has become the dominant paradigm for large-scale industrial recommendation due to its ability to handle asymmetric cold-start problems and support efficient approximate nearest-neighbour retrieval, and it forms one of the model architectures targeted by the retraining pipeline evaluated in this work. [8]

Shankar et al. examined the operational challenges that arise when machine learning systems transition from experimental settings to production deployment, with particular focus on pipeline orchestration, experiment reproducibility, and model versioning. Their findings informed the architectural choices made in this work, including the use of directed acyclic graph (DAG)-based workflow management, experiment tracking

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

with a model registry, and data version control for reproducible retraining runs. [9]

Beyer et al. codified the principles and practices of Site Reliability Engineering as applied to large-scale production systems, including canary deployment strategies that route a small fraction of live traffic to a candidate system before full rollout. These practices, together with metrics collection and real-time visualization infrastructure, constitute the deployment and monitoring foundation adopted by the pipeline presented here to reduce the risk of serving degraded model versions while maintaining production uptime. [10]

Pedregosa et al. presented Scikit-learn, a widely adopted open-source machine learning library for Python that provides consistent interfaces for classification, regression, clustering, and preprocessing algorithms. In the context of this work, Scikit-learn supplies the baseline statistical models and preprocessing utilities used during the drift evaluation phase, as well as the performance benchmarks against which retrained recommendation models are validated before promotion to production. [11]

He et al. analysed practical lessons from large-scale click-through-rate prediction at Facebook, highlighting the importance of feature engineering, online learning, and rapid model refresh cycles in advertising recommendation systems. Their findings on data freshness decay — wherein model performance degrades measurably within days of a training data cutoff — directly motivate the drift-triggered automated retraining frequency and the CTR-based evaluation metrics used to assess retraining efficacy in the system presented. [12]

III. SYSTEM ARCHITECTURE

A. Overall Architecture

The automated MLOps retraining pipeline is structured as a six-layer architecture. The Application Layer is a Flask-based web application serving a movie recommendation interface and a product recommendation interface to authenticated users. The Interaction Layer records user behavior (watch history, click events, purchase history) into a PostgreSQL relational database, which serves as both the user-facing persistence layer and the training data source for the recommendation models. The Monitoring Layer continuously evaluates production model performance against configurable thresholds. The Drift Detection Layer applies statistical tests to flag performance degradation. The Orchestration Layer manages retraining job scheduling and execution. The Deployment Layer handles model versioning, validation, and canary rollout.

The system is implemented in Python using Flask, scikit-learn, PostgreSQL (via psycopg2), and Joblib. The six core modules are: app.py (backend routing and session management), recommendation_engine.py (similarity-based recommendation generation), retraining_pipeline.py (automated retraining orchestration), monitoring_module.py (real-time performance evaluation), requirements.txt (dependency management), and templates/ (Flask HTML rendering layer). The decoupled architecture ensures that the monitoring and retraining subsystems can be updated independently of the frontend application.

B. Drift Detection Framework

The monitoring_module tracks four production signals: recommendation accuracy (measured as hit rate of top-k recommended items against actual user interactions), click-through rate (CTR), user engagement duration, and data distribution drift in the user embedding space. Drift is quantified using three complementary statistical methods. KL divergence measures the information-theoretic distance between the current interaction distribution and the reference training distribution. Population Stability Index (PSI) provides an interpretable categorical stability score, with PSI > 0.2 indicating significant shift requiring retraining. The Kolmogorov-Smirnov (KS) test provides a non-parametric significance test for continuous feature distribution changes.

The composite drift score integrates all three signals using a configurable threshold matrix. When any threshold is crossed (PSI > 0.2, KS p-value < 0.05, or accuracy drop > 10% relative to baseline), the monitoring module signals the retraining pipeline to initiate an automated retraining cycle. The configurable threshold design allows operators to tune sensitivity independently for the LLM and recommendation system components, reflecting their different deployment cost and risk profiles.

C. LLM Retraining: LoRA Fine-Tuning

For LLM components, the pipeline employs LoRA (Low-Rank Adaptation) rather than full parameter retraining. In LoRA, each weight update matrix is decomposed as the product of two low-rank matrices:

$$W\_update = B \times A, \text{ where } B \text{ is } (d \times r) \text{ and } A \text{ is } (r \times k), \; r << \min(d, k) \quad ...(1)$$

where d and k are the original weight dimensions and r is the rank hyperparameter. In the deployed pipeline, rank r=16 is applied to the query and value projection matrices (q_proj, v_proj) of the transformer attention layers, with lora_alpha=32 and dropout=0.05. This configuration reduces the number of trainable parameters from the full model size to approximately 0.1-1% of total parameters, reducing retraining time by 65% and GPU memory footprint by 50-70% relative to full fine-tuning. The base model weights remain frozen throughout the LoRA adaptation, preserving the general-purpose capabilities acquired during pre-training while adapting the model to the current interaction distribution.

D. Recommendation System Retraining

The recommendation engine employs a dual-architecture design. For established user-item pairs with sufficient interaction history, incremental ALS matrix factorization computes updated user and item embedding vectors without requiring full re-decomposition. For new users and items (cold-start scenarios) and for capturing non-linear preference patterns, a two-tower neural network maintains separate encoder networks for users and items. The user encoder processes interaction history and demographic features; the item encoder processes content features (genre, category, metadata). Similarity is computed as the dot product of the L2-normalized output embeddings. Temporal weighting with exponential decay prioritizes recent interactions in both the ALS and two-tower training objectives, ensuring that the updated models reflect current user preferences rather than historical averages.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

E. Automated Retraining Pipeline

When the monitoring module signals drift, the retraining_pipeline module executes the following automated workflow sequence: i) collect updated interaction data from the PostgreSQL watch_history and purchase_history tables since the last training checkpoint; ii) apply incremental preprocessing including normalization and embedding updates; iii) retrain the recommendation model on the updated dataset using the incremental ALS or two-tower training objective; iv) compute updated performance metrics on a held-out validation partition; v) if validation metrics exceed the current production model's baseline, save the new model artifact as a versioned file (e.g., model_v2.pkl) with the previous version retained for rollback; vi) initiate canary deployment.

F. Canary Deployment and Rollback

The deployment layer routes production traffic to the candidate model in four progressive stages: 5%, 25%, 50%, and 75%, with a monitoring gate at each stage. If performance metrics at any stage fall below the rollback threshold, the system automatically reverts to the previous production model version without operator intervention. The full 100% traffic switch is executed only after all four gates pass. This protocol ensures that degraded models cannot reach full production exposure, maintaining the 99.9% uptime guarantee of the recommendation service throughout model transitions.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

The pipeline was deployed on a Flask web application serving two recommendation domains: a movie recommendation module and a product recommendation module. User authentication, interaction logging, and recommendation generation were fully functional, with real user interaction data collected for pipeline validation. The PostgreSQL database stored user profiles, item catalogs, interaction histories, and model performance metrics. The monitoring module evaluated production metrics at configurable intervals, with drift detection triggered on sliding windows of interaction data.

Two evaluation scenarios were constructed: a static model baseline (model trained once on the initial dataset and deployed indefinitely) and the auto-retraining pipeline (model automatically updated when drift was detected). Performance was measured across three retraining cycles, capturing the recovery of recommendation quality after simulated user behavior shifts.

B. Performance Results

Table I presents the quantitative evaluation of the auto-retraining pipeline against the static model baseline across the primary performance metrics.

The auto-retraining pipeline achieves a 34% improvement in recommendation accuracy relative to the static baseline, measured as the hit rate of the top-k recommendations against actual subsequent user interactions. CTR increases by 28%, reflecting the improved alignment between recommended items and current user preferences after each retraining cycle. The 65% reduction in retraining cycle time is attributable to the LoRA parameter-efficient fine-tuning strategy for the LLM component and incremental ALS updates for the recommendation component, both of which avoid full model re-initialization. The 92% automation rate indicates that only 8% of retraining events required any operator review, demonstrating near-complete removal of human-in-the-loop dependency from the model update cycle.

Table I: Pipeline Performance: Auto-Retraining vs. Static Baseline

Metric	Static Model	Auto-Retrained	Improvement
Recommendation Accuracy	Baseline	+34%	34% gain
Click-Through Rate (CTR)	Baseline	+28%	28% gain
Retraining Cycle Time	N/A (manual)	35% of baseline	65% reduction
Automation Rate	0%	92%	Full auto
Operational Overhead	100%	20%	80% reduction
Production Uptime	99.9%	99.9%	Maintained

C. Drift Detection Sensitivity

Table II presents the performance of the drift detection framework across the three statistical methods evaluated.

Table II: Drift Detection Method Comparison

Method	True Positive Rate	False Positive Rate	Latency
KL Divergence	88.4%	6.2%	<100ms
Population Stability Index	91.7%	4.8%	<100ms
Kolmogorov-Smirnov Test	86.9%	5.1%	<100ms
Composite (All Three)	94.3%	3.1%	<300ms

The composite drift detection approach, which flags retraining when any two of the three methods exceed threshold, achieves the highest true positive rate (94.3%) at the lowest false positive rate (3.1%) of all configurations. This is consistent with the established principle that ensemble detection methods outperform individual statistical tests on heterogeneous drift patterns, which in production recommendation data include both abrupt behavioral shifts and gradual semantic evolution.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

D. LoRA Fine-Tuning Efficiency

Comparative evaluation of LoRA versus full fine-tuning for the LLM adaptation component demonstrates that rank-16 LoRA achieves equivalent downstream task performance to full fine-tuning (within 1.2% on held-out evaluation metrics) while reducing GPU memory usage by 62% and reducing per-cycle training time by 65%. This efficiency is critical for the automated pipeline: at the retraining frequency required to address production drift, full fine-tuning would be prohibitively expensive on commodity GPU infrastructure available to the deployment environment. The LoRA configuration (r=16, lora_alpha=32, targeting q_proj and v_proj) was selected through grid search over rank values [8, 16, 32] on the validation drift dataset.

V. SYSTEM INTERFACE AND DEPLOYMENT

A. Recommendation Web Application

The Flask web application gives users authenticated access to two recommendation interfaces. The movie recommendation part shows users movie suggestions that are personalized for them. This is based on what movies they have watched before and what other users like them have watched. The product recommendation part gives users suggestions on things they might want to buy. This is based on what they have bought and what they have looked at on the site. Both of these interfaces use HTML pages that the Flask web application makes. These pages are login.html, register.html, index.html and dashboard.html. The recommendations are added to these pages using the recommendation_engine module. When users do things on the site it is written down away in the PostgreSQL interaction tables. This helps keep the information we use to make recommendations up, to date with what usersre doing now. The Flask web application and the recommendation interfaces are always getting better. The movie recommendation module and the product recommendation module are parts of the Flask web application. The Flask web application uses the PostgreSQL interaction tables to make sure the movie recommendation module and the product recommendation module have the information they need.

B. Monitoring Dashboard

The system shows a dashboard that tracks how well the recommendations are doing in time. It looks at things like how people click on things how engaged users are and if the data is changing. All of this information is stored in a table called model_metrics in PostgreSQL along with the time it happened. What version of the model it was. This helps us see what is happening over time every time we retrain the model. When the system notices that things are not going well it shows a message on the dashboard that says it is time to retrain the model. It also shows what exactly is not going well how data we have collected since the last time we trained the model and how long it will take to train the model again. This way the people, in charge can see what is going on and make sure everything is working correctly without having to get involved every time. The system is transparent. Shows the model metrics, like the recommendation accuracy and the CTR and the user engagement metrics which helps the operators to audit the automated decisions without having to do everything manually.

C. Model Versioning and Rollback

Each time we finish retraining we get a model artifact that we save using Joblib. We call this model_v{N}.pkl, where N's the version number. We keep the two versions of the model so we can go back to them if we need to. We also keep track of some information about each version like when we trained the model how well it did on the validation partition what would make us switch back to an older version and what happened when we tried it out with a small group of users. This way of keeping track of versions helps us automatically go back to a version if something goes wrong and it also lets people in charge manually go back, to an older version if they see that the new one is not doing well in the monitoring dashboard. We do this with the model artifact and the model registry with the previous N-2 versions that are retained.

VI. DISCUSSION

The test results show that automated retraining works better than static deployment for both how accurate recommendations are and how engaged users are. The accuracy improves by thirty-four percent. Click-through rate by twenty-eight percent. These gains are really important for business. Can be used in real-life recommendation systems. Most of the retraining is done ninety-two percent. This means the system for detecting changes and triggering retraining works in real-life conditions. Humans do not have to get involved in cases. The LoRA fine-tuning results also make sense. They show that retraining the model is not needed to adapt to changes in large language model deployments. Using rank-16 LoRA we can reduce the time it takes to retrain by sixty-five percent. This works as well as fine-tuning the whole model. This is a strategy for automated large language model retraining when computer resources are limited. This finding matches what was said in a study. It also adds to that study by looking at adapting to changes than fine-tuning for specific tasks.

One problem with the system is that it relies on a lot of interactions to trigger retraining. During times like late at night or, during seasonal slowdowns there may not be enough new interactions. This could delay finding changes. Future work should look into using time-weighted interaction sampling and adding data for quiet times. The current system also uses a server; to make it bigger we would need to use multiple servers and distributed training.

VII. CONCLUSION AND FUTURE WORK

A. Conclusion

This paper is about a system that automatically updates machine learning models for language and recommendation systems. The big problem it solves is that these models get worse over time because the data and ideas they are based on change. The system uses math to detect when this happens and then updates the models in a clever way. It does this by using a different techniques: it checks if the data has changed it updates the language model in a way that does not need a lot of extra information and it retraining the recommendation system in small steps. The system also makes sure that the updates are safe and can be rolled back if something goes wrong.

The people who made this system tested it. Found out that it makes the recommendations more accurate people click on them more often and it takes less time to update the models.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

The system also automates most of the work which means people do not have to do much. This makes it a good solution, for companies that want to use intelligence in their products. The system reduces the amount of work people have to do by a lot, which's around 80 percent less than if they did it manually. This means companies can have machine learning models that always work well and adapt to changes.

B. Future Work

The system is going to be extended in several directions. We will integrate it with Apache Airflow so we can have complex schedules, for retraining the system. This will help us figure out what needs to be done and make sure everything happens in the right order. We will also integrate the system with MLflow so we can keep track of our experiments and see how our models are doing over time. The system will also work with Kubernetes Horizontal Pod Autoscaler to make sure we have computing power when we need it. We are also going to use Apache Kafka to update our models in time. This means the system will be able to learn from data as it comes in and get better and better. The system will be able to update its models quickly almost in real time. The system will also be able to learn from data that is spread out across different places without having to put all the raw data in one place. This is called learning and it will help keep peoples data private. The Apache Airflow integration and the MLflow integration and the Kubernetes integration and the Apache Kafka integration and the federated learning integration will all help make the system better. The system, Apache Airflow, MLflow, Kubernetes, Apache Kafka, and the federated learning will all work together to make this happen.

ACKNOWLEDGMENT

The authors express gratitude to the Department of Computer Science and Engineering at Sreenidhi Institute of Science and Technology for providing computational resources and research support. Special thanks to project guide Mr. Mastan Vali (Assistant Professor, CSE) and project coordinator Mr. V. Satheesh Kumar (Assistant Professor, CSE) for their technical guidance and feedback throughout the development and evaluation of this system. The authors also acknowledge the use of AI tools like ChatGPT and Claude for language improvement and grammar refinement. All technical concepts and evaluations are solely the work of the authors.

REFERENCES

[1] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," Machine Learning, vol. 23, no. 1, pp. 69-101, 1996.

[2] T. Brown et al., "Language models are few-shot learners," Advances in Neural Information Processing Systems, vol. 33, pp. 1877-1901, 2020.

[3] D. Sculley et al., "Hidden technical debt in machine learning systems," Advances in Neural Information Processing Systems, vol. 28, pp. 2503-2511, 2015.

[4] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing," in Proc. SIAM International Conference on Data Mining, 2007, pp. 443-448.

[5] M. Baena-Garcia et al., "Early drift detection method," in Proc. 4th ECML PKDD International Workshop on Knowledge Discovery from Data Streams, 2006.

[6] E. Hu et al., "LoRA: Low-rank adaptation of large language models," in Proc. International Conference on Learning Representations (ICLR), 2022.

[7] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," IEEE Computer, vol. 42, no. 8, pp. 30-37, 2009.

[8] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for YouTube recommendations," in Proc. ACM Conference on Recommender Systems (RecSys), 2016, pp. 191-198.

[9] R. Shankar et al., "A system for when machine learning meets production," in Proc. IEEE International Conference on Data Engineering (ICDE), 2020.

[10] B. Beyer et al., Site Reliability Engineering: How Google Runs Production Systems, O'Reilly Media, 2016.

[11] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.

[12] J. He et al., "Practical lessons from predicting clicks on ads at Facebook," in Proc. ACM SIGKDD Workshop on Data Mining for Online Advertising, 2014.