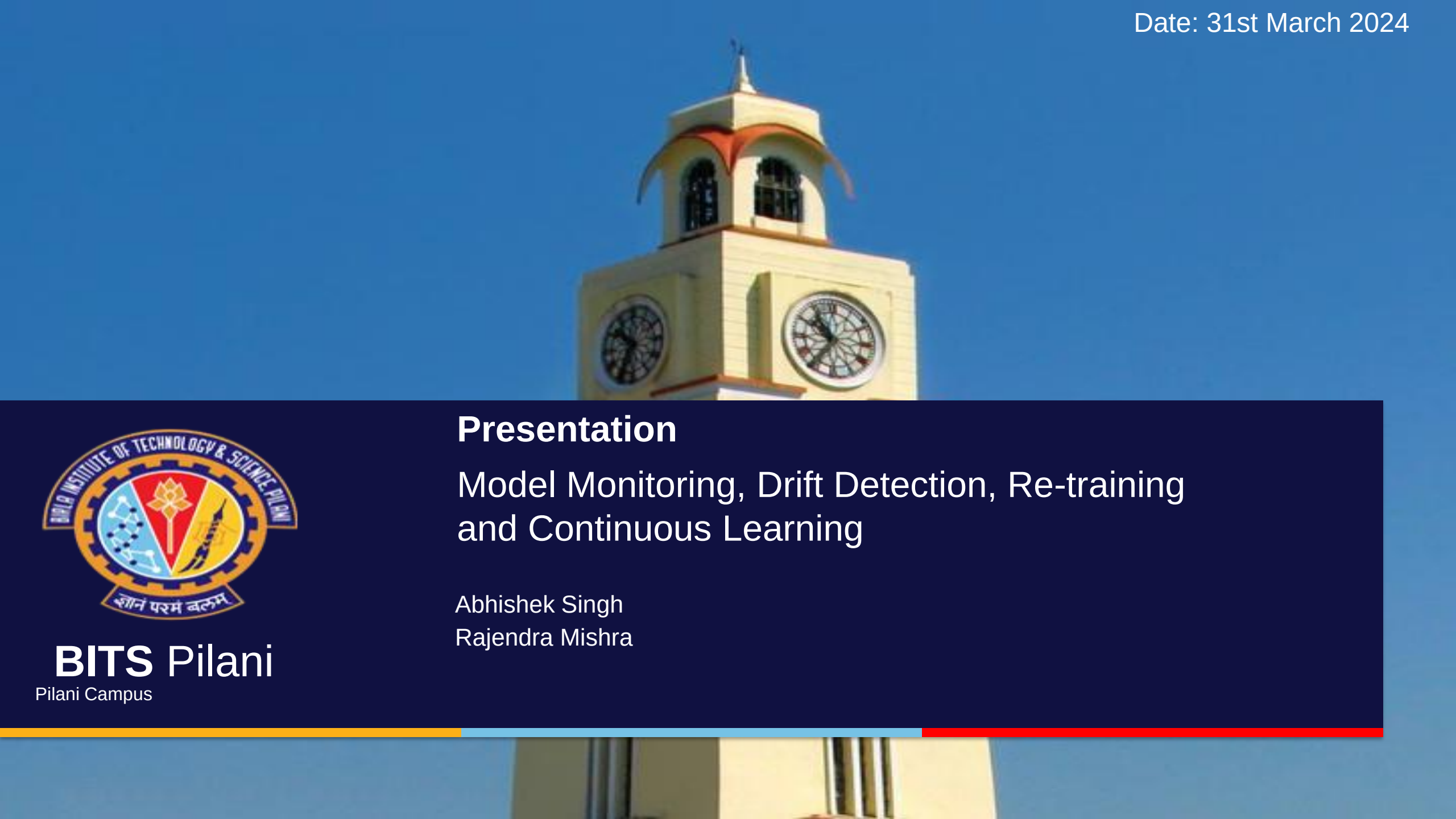
Date: 31st March 2024



Presentation

Model Monitoring, Drift Detection, Re-training and Continuous Learning

Abhishek Singh
Rajendra Mishra

BITS Pilani
Pilani Campus

Agenda

- Introduction
- Dashboard Tools
- Infrastructure monitoring
- ML Model Monitoring
- Model Training Monitoring
- Model Tuning Monitoring
- Model Serving resources monitoring
- Model Serving Monitoring
- Model re-training and deployment
- Model Re-training Policy
- Stateless Vs Stateful model re-training
- Four Stages of Continuous Learning
- Summary
- Python Packages available to detect drifts
- References
- Demo

Introduction

In machine learning system lifecycle, an important step begins, once we deploy our models to production.

Traditionally, with rule based, deterministic, software, the majority of the work occurs at the initial stage and once deployed, our system works as we've defined it.

But with machine learning, we haven't explicitly defined how something works but used data to architect a ***probabilistic solution***.

This approach is subject to **natural performance degradation over time**, as well as unintended behaviour**, since gradually the data exposed to the model will be different from what it has been trained on**. This isn't something we should be trying to avoid but rather **understand and mitigate as much as possible**.

In addition, there are **many aspects of the machine learning system that we need to monitor** to ensure that our machine learning use cases works as expected.

•Infrastructure monitoring

•ML Model Monitoring
1. Model Training Monitoring
2. Model Serving Monitoring

•Model Serving resources monitoring

Dashboard Tools

Dashboard tools play a crucial role in the MLOps lifecycle by providing insights into model performance, detecting potential issues, and ensuring that machine learning systems remain robust and efficient.

Prometheus

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is widely used for monitoring both machine learning models and infrastructure components, making it an essential tool for MLOps. Key features of Prometheus include:

•Flexible data model and powerful query language (PromQL), enabling users to define custom metrics and perform complex queries to gain insights into their systems
•Pull-based data collection, allowing Prometheus to scrape metrics from multiple sources, such as applications, databases, and other infrastructure components
•Built-in alerting system, enabling users to define custom alert rules and receive notifications when specific conditions are met
•Integration with popular visualization tools, such as Grafana, providing users with customizable dashboards and visualizations of their monitoring data

By incorporating Prometheus into their MLOps workflows, data scientists and engineers can effectively monitor their machine learning models and infrastructure components, ensuring the reliability and performance of their systems.

Dashboard Tools

Grafana

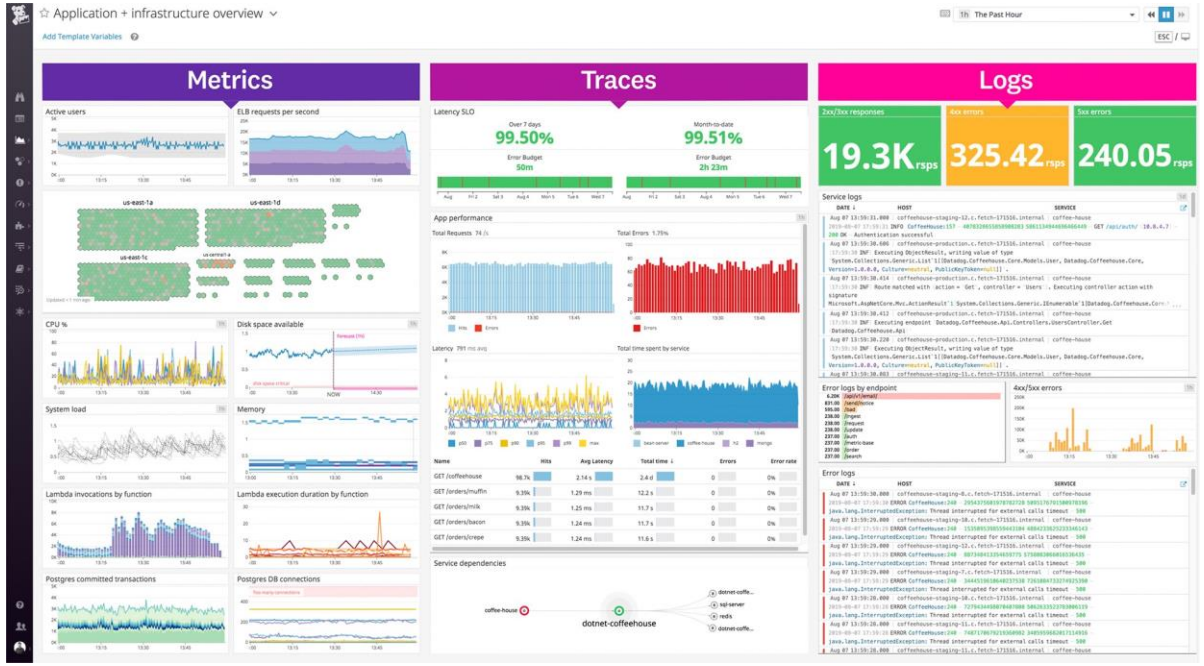
Grafana is an open-source analytics and monitoring platform that allows users to create, explore, and share dashboards and visualizations of their data. Grafana supports a wide range of data sources, including Prometheus, Elasticsearch, and InfluxDB, making it a versatile option for monitoring and performance management in MLOps. Key features of Grafana include:

•Customizable dashboards, allowing users to create and share visualizations of their monitoring data, such as model performance metrics and infrastructure metrics
•Flexible alerting system, enabling users to define custom alert rules and receive notifications through various channels, such as email, Slack, or PagerDuty
•Extensive plugin ecosystem, providing users with additional data sources, panels, and themes to customize their monitoring experience
•Integration with popular authentication and authorization systems, such as OAuth, LDAP, and GitHub, ensuring secure access to monitoring data

By using Grafana in their MLOps workflows, data scientists and engineers can gain valuable insights into their models' performance and infrastructure, identify potential issues, and optimize their systems for better results

Infrastructure monitoring

The first step to ensure that our model is performing well is to ensure that the actual system is up and running as it should. This can include metrics specific to service requests such as latency, throughput, error rates, etc. as well as infrastructure utilization such as CPU/GPU utilization, memory, etc. Fortunately, most MLOps providers and even CaaS layers will provide this insight into our system's health through a dashboard. In the event we don't, we can easily use Grafana, Datadog, etc. to ingest system performance metrics from logs to create a customized dashboard and set alerts. (we need to rely on the underlying platform for this)



ML Model Monitoring

Just monitoring the system's health won't be enough to capture the underlying issues with our model.

So we need various levels of model monitoring.

We need to **monitor training and tuning metrics**, **monitoring the model serving resources** e.g. throughput & latency, and finally, **model monitoring for issues like data drift, model decay** to take a decision to trigger re-training of the model in case the prediction drifts from ground truth beyond a threshold.

Model Training Monitoring

The first level of metrics to monitor and log involves the model training and tuning performance. These would be quantitative evaluation metrics that we used during model training and evaluation e.g. accuracy, precision, f1, etc. There are different metrics to be evaluated in case of regression or classification cases.

Regression Metrics
The most common metrics for evaluating predictions on regression machine learning problems:

•Mean Absolute Error.
•Mean Squared Error.
•R^2.
•Training parameters

Classification Metrics
Classification problems are perhaps the most common type of machine learning problem and as such there are a lots of metrics that can be used to evaluate predictions for these problems:

•Classification Accuracy.
•Logarithmic Loss.
•Area Under ROC Curve.
•Confusion Matrix.
•Classification Report.
•Training parameters

Model Tuning Monitoring

Once we've done model training and evaluation, it's possible that we want to see if you can further improve our training in any way. We can do this by **tuning hyper-parameters**. There were a few parameters we implicitly assumed when we did our training, and now is a good time to go back and test those assumptions and try other values. the following parameters can be monitored and logged:

- Fine-tuning parameters
- Metric related to model accuracy

In machine learning, the terms **model parameters** and **hyperparameters** are often used interchangeably. However, there is a subtle difference between the two.

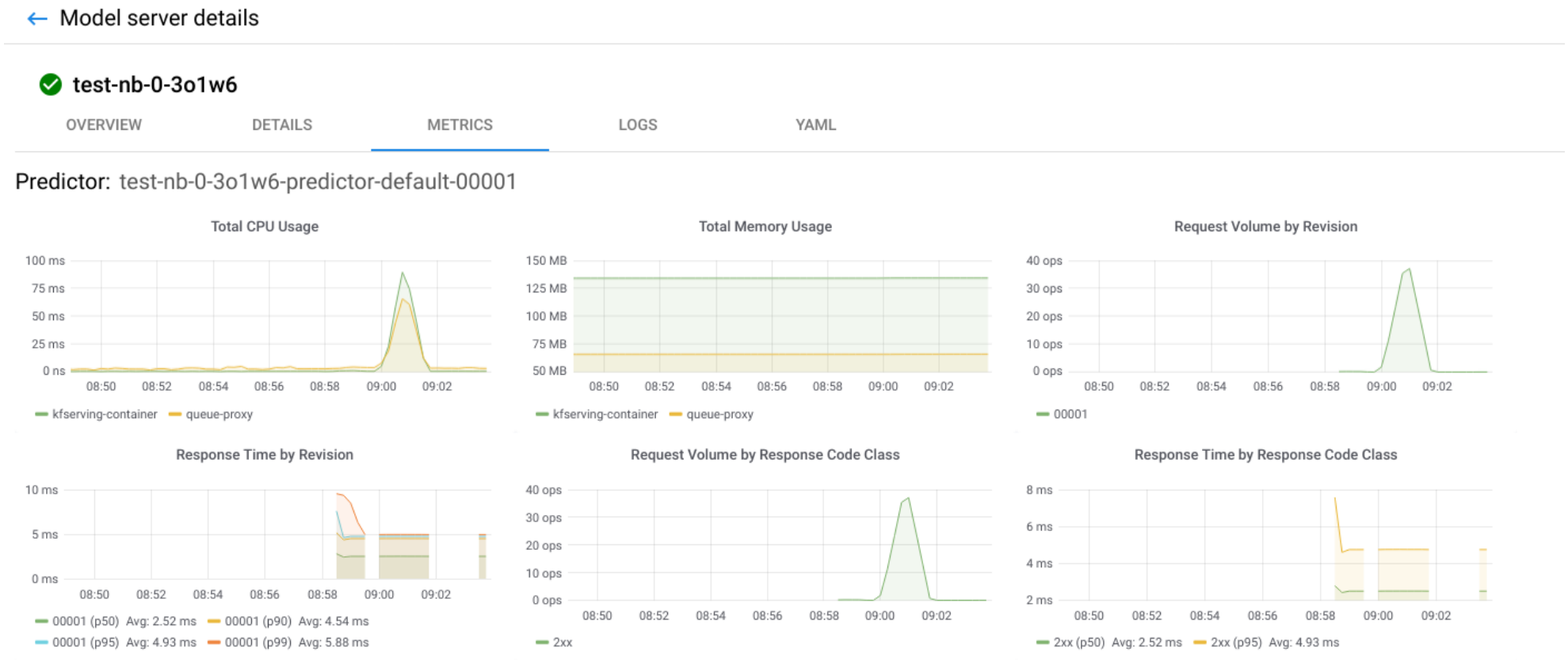
•**Model parameters** are the values that are learned by the model during training. They are typically represented by a vector of numbers. The number of model parameters depends on the complexity of the model. For example, a simple linear regression model may have only a few model parameters, while a complex deep learning model may have millions or even billions of model parameters.

•**Hyperparameters** are the values that are set before training the model. They control the learning process and the behavior of the model. For example, a hyperparameter might be the number of iterations to train the model, the learning rate, or the batch size. Hyperparameters can have a significant impact on the performance of the model.

In general, model parameters are learned from data, while hyperparameters are set by the user. However, there are some cases where model parameters can be set by the user, such as when using a pre-trained model.

Model Serving resources monitoring

An important part of monitoring system resources is monitoring the performance of model serving infrastructure. Once the model is trained and served, we are going to use the model serving for quiet sometime. So its important to ensure that the performance of model serving is as per expectation..



Model Serving Monitoring

There are 2 questions which we need to answer:

1. Does new incoming data reflect the same patterns as the data on which model was originally trained on ?
2. Is the model performing as well in development as it did in design phase ? If not, why ?

Machine Learning Model Drift

How well a model performs is the reflection of the data used to train it. If there is a significant change in distribution or composition of values of input variables or the target variables, it could lead to data drift which could cause the model to degrade. Model degradation can lead to inaccurate predictions and insights, therefore we want to carefully monitor model degradation. *To track model degradation there are 2 approaches to consider:*

1. ***Based on Ground Truth*** - Labelled data is compared to prediction
2. ***Based on Data Drift*** - Instead of waiting for the labelled data, the training data and recent data are compared statistically.

Model Serving Monitoring

Model Monitoring based on Ground Truth vs Data Drift

The ground truth is the correct answer that the model was asked to solve -- when we know the ground truth for all the predictions a model has made, we can judge with certainty how well the model is performing.

Ground truth monitoring requires waiting for the label event and then computing the performance of the model based on these ground truth observations. When the ground truth is available quickly it can the best solution to monitor model degradation, however obtaining ground truth can be slow and costly.

If our use cases requires rapid feedback, or if the ground truth is not available or hard to compute, input drift evaluation may be the way to go. The basis for input drift, is that the model is only going to predict accurately , if the data it was trained on is an accurate reflection of the real world, so if a comparison of the recent requests to a deployed model against the training data , shows distinct differences, then there is a strong likelihood that the model performance may be compromised.

Unlike for ground truth evaluation, the data required for input drift evaluation already exists, so there is no need to wait for any other information.

Note: ***If we wait to catch the model decay based on the ground-truth performance, it may have already caused significant damage to downstream business pipelines that are dependent on it. We need to employ more fine-grained monitoring to identify the sources of model drift prior to actual performance degradation.***

Model Serving Monitoring

Types of Drift

We need to first understand the different types of issues that can cause our model's performance to decay (model drift). The best way to do this is to look at all the moving pieces of what we're trying to model and how each one can experience drift.

Drift Type	Description
Feature/Input Drift → (X)	Input feature(s) distributions deviate. Also known as ***Covariate Shift.***
Label/Target Drift → (y)	Label/Target distribution deviates. Also known as ***Prior Probability Shift.***
Prediction Drift → P(y)	Model prediction distribution deviates. Also known as ***Model Drift***
Concept Drift → P(y|X)	External factors cause labels to evolve. Also known as ***Task Drift***

Model Serving Monitoring

Drift types and actions to take:

Feature/Data Drift	• Investigate feature generation process • Retrain using new data
Label/Target Drift	• Investigate label generation process • Retrain using new data
Prediction Drift	• Investigate Model training process • Access business impact of changes in prediction
Concept Drift	• Investigate additional feature engineering • Consider alternative approach/solution • Retrain/tune using new data

Model re-training and deployment

•It's usually a good idea to establish policies around when we're going to retrain our model.

•There's no right or wrong answer here, so it will depend on what works in our particular situation.

•we could simply choose to retrain our model whenever it seems to be necessary.

•That includes situations where we detect a drift, but it also includes situations where we always retrain the models according to a schedule, or on demand.

•***In practice, this is what many companies do, because it's simple to understand and in many domains, it works fairly well.***

•It can, however, incur higher training and data gathering costs unnecessarily.

•If we can automate the process of detecting the conditions which require model retraining, that will be ideal.

•That includes being able to detect model performance degradation or when we detect significant data drift, and triggering retraining.

•In both cases, in order to automate retraining, we should have data gathered and labelled automatically and only retrain when sufficient data is available.

•We need to have continuous training, integration and deployment setup to make the process fully automated.

Model Re-training Policy

Policy	Intuition
On Demand	Re-train the model on need basis (e.g. when ACTS detects that the deployed model is old, it triggers re-training and deployment)
On Schedule	Re-train at a fixed internal on a daily, weekly or monthly basis.
On Data Drift	We find significant changes in the input data and trigger model retraining.
On Model Performance Degradation	Prediction Drift based on ground truth, trigger model retraining.
On New data availability	When new data is available on ad-hoc basis and there is no trend. Re-train model when new labelled data is collected and available in source database.

Stateless Vs Stateful model re-training

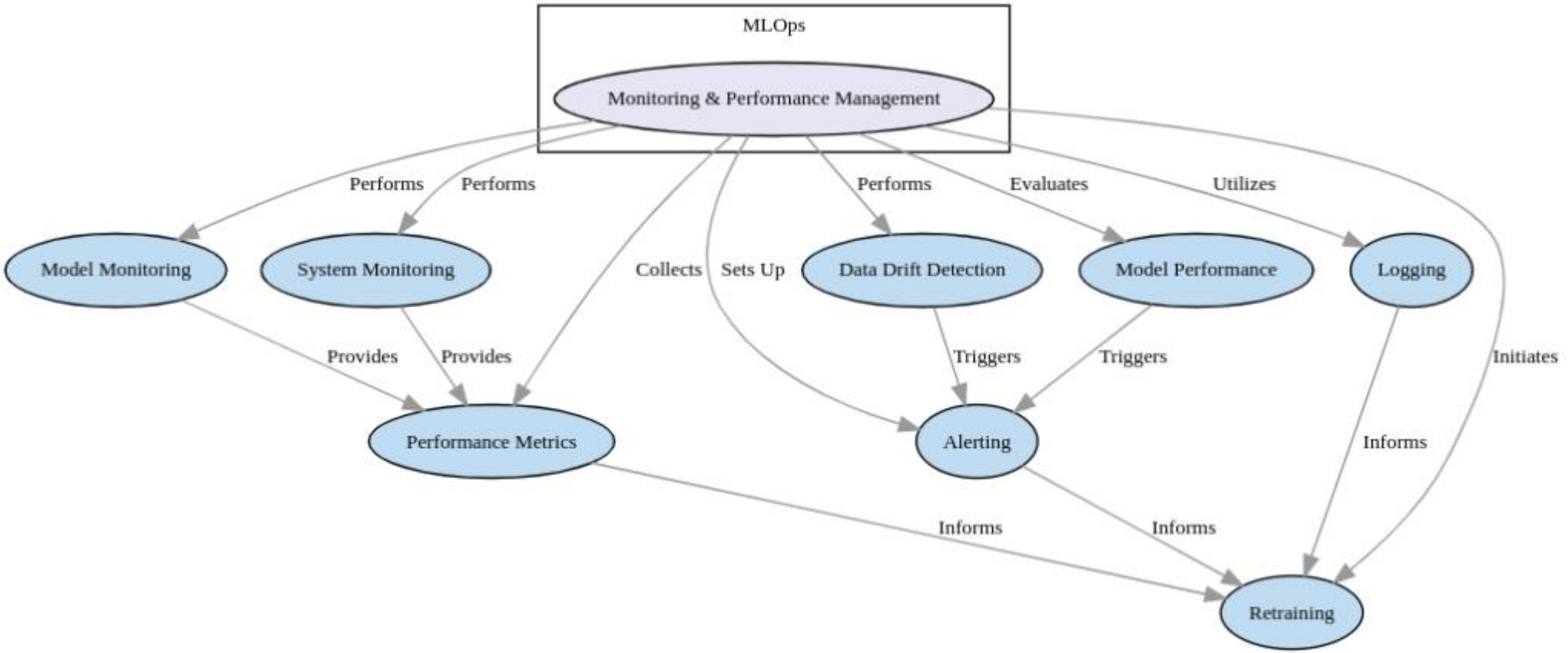
Policy	Intuition
Stateless	• New model is retrained from scratch each time.
Stateful	• The existing model continues to be trained on new data • Stateful re-training is also known as fine-tuning or incremental learning

Four Stages of Continuous Learning

innovate achieve lead

Stage	Description
Stage 1: On-Demand, Stateless retraining	This is the first stage of continuous learning. In this stage we update the models on demand. In this stage the models are updated on need basis or as deemed suitable.
Stage 2: Automated, Stateless retraining	In this stage we write automated and recurring pipelines which rely mostly on schedule to trigger re-training. Most companies with somewhat mature ML infrastructure are in this stage.
Stage 3: Automated, Stateful retraining	In this stage we update existing model which continues to be trained on new data. Stateful re-training is also known as fine-tuning or incremental learning.
Stage 4: Continual learning	Till Stage 3 our models are still updated based on a fixed schedule. This might not be optimal always as quite a few models might not show any performance degradation and hence the stage 3 might lead to higher compute costs. On the flip side some of our models could decay/degrade much faster and require a much faster and automated retraining. In stage 4 our re-training trigger can be combination of one or more factors e.g. data drift, model performance degradation (beyond threshold) or arrival of new data.

Summary


MLOps
Monitoring & Performance Management
Performs
Performs
Performs
Evaluates
Utilizes
Model Monitoring
System Monitoring
Collects
Sets Up
Data Drift Detection
Model Performance
Logging
Provides
Provides
Triggers
Triggers
Initiates
Performance Metrics
Alerting
Informs
Informs
Informs
Retraining

Python Packages available to detect drifts

Some of the open-source packages available to detect drift are:

Evidently AI: Open-Source Tool To Analyze Data Drift. Evidently is an open-source Python library for data scientists and ML engineers. It helps evaluate, test, and monitor the performance of ML models from validation to production.

drift_detection: This package contains some developmental tools to detect and compare statistical differences between 2 structurally similar pandas dataframes. The intended purpose is to detect data drift — where the statistical properties of an input variable change over time. It provides a class DataDriftDetector which takes in 2 pandas dataframes and provides a few useful methods to compare and analyze the differences between the 2 datasets.

skmultiflow: One of the key features of skmultiflow is its ability to handle concept drift, which is the change in the underlying distribution of the data over time. It includes algorithms for detecting concept drift and adapting the machine learning model in real-time to account for the changes in the data. This makes it an ideal tool for building machine learning models that can adapt and continue to perform well as the data changes over time.

Deepchecks: Deepchecks Open Source is a python library for data scientists and ML engineers. The package includes extensive test suites for machine learning models and data, built in a way that's flexible, extendable, and editable.

NannyML: NannyML is an open-source Python library that helps you monitor and improve the performance of your machine learning models

References

- An overview of unsupervised drift detection methods
- Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift
- Monitoring and explainability of models in production
- Detecting and Correcting for Label Shift with Black Box Predictors
- Outlier and anomaly pattern detection on data streams

Thank You !

Demo