Introduction to MLOps

Mohamed ASSELLAOU

AGENDA

1. MLOps Fundamentals
2. Key Concepts
3. Feature Pipelines
4. Training Pipelines
5. Inference Pipelines
6. Deployment & Monitoring

MLOps overview

The bridge

MLOps connects machine learning development with operational systems. It ensures models work in real-world environments.

Traditional failures

Most ML workflows fail in production due to inconsistent environments, manual processes, and poor monitoring.

DevOps evolution

MLOps builds on DevOps principles while addressing unique challenges of machine learning systems.

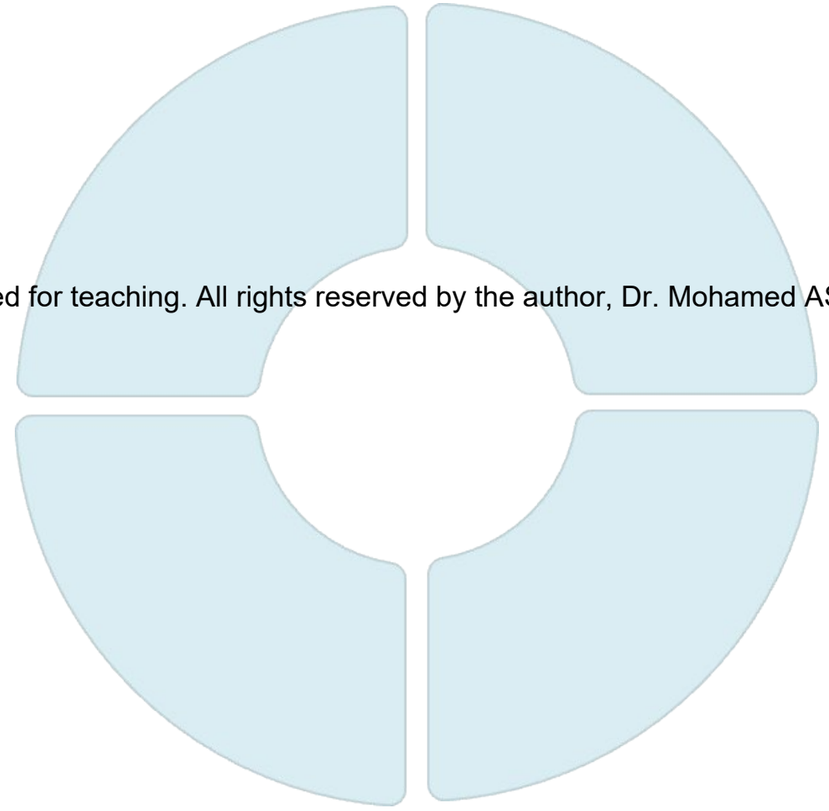
What is MLOps?

Machine learning

Algorithms, models, and experimentation

DevOps

Automation, CI/CD, infrastructure



Goal

Reliable and efficient ML lifecycle management

Data engineering

Data pipelines, validation, storage

Why MLOps matters

90%

Failed projects

Percentage of ML projects that never reach production

10x

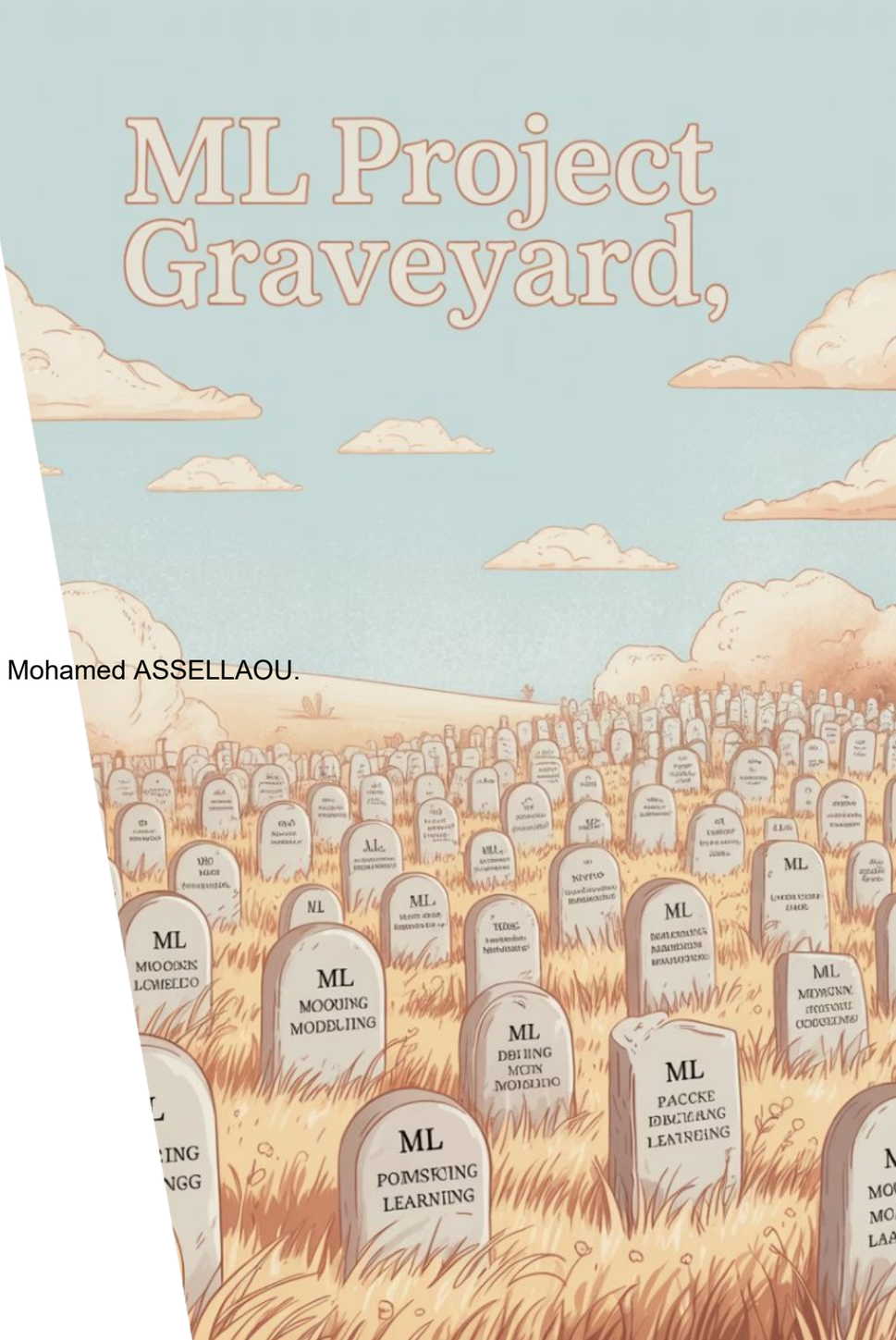
Technical debt

ML systems accumulate technical debt faster than traditional software

40%

Performance drop

Average model accuracy decrease due to data drift in 3 months



The ML lifecycle challenge

System complexity

ML systems have more moving parts than traditional software. They combine code, data, and models.

Data dependencies

Changing data creates new failure points. Models depend on specific data distributions.

Monitoring needs

Models require continuous evaluation. Performance degrades over time without retraining.

Team coordination

Cross-functional teams need shared workflows. Data scientists and engineers must collaborate effectively.

MLOps maturity levels

Level 3: Full Automation

Automated end-to-end MLOps lifecycle

Level 2: CI/CD Automation

Automated testing and deployment

Level 1: ML Pipeline Automation

Automated training workflows

Level 0: Manual Process

Manual experimentation and deployment

MLOps core principles

Version control

Track all components: code, data, models, and configurations. Nothing exists without versioning.

Continuous integration

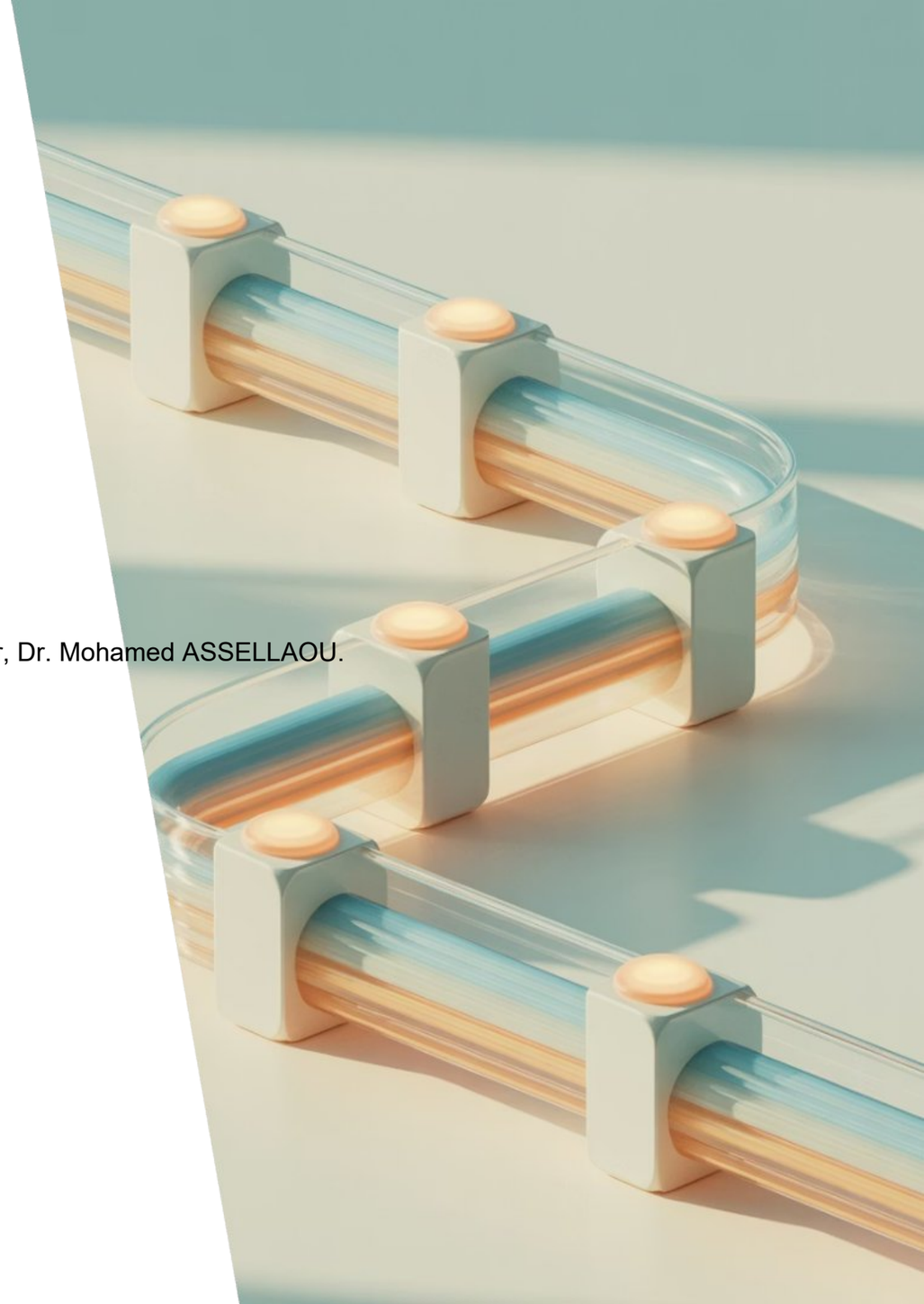
Automatically test ML components when changes occur. Validate code, data, and models.

Continuous delivery

Automate model deployment to target environments. Ensure consistency and reliability.

Continuous training

Automatically retrain models on new data. Monitor for drift and performance degradation.



Evolution of ML Systems

Ad-hoc notebooks

Manual experimentation and deployment. No standardization or reproducibility. High technical debt.

Monolithic pipelines

Single end-to-end pipelines. Difficult to maintain and test. Tight coupling between components.

Modern FTI approach

Separate Feature/Training/Inference pipelines. Clear interfaces. Independent development and operation.

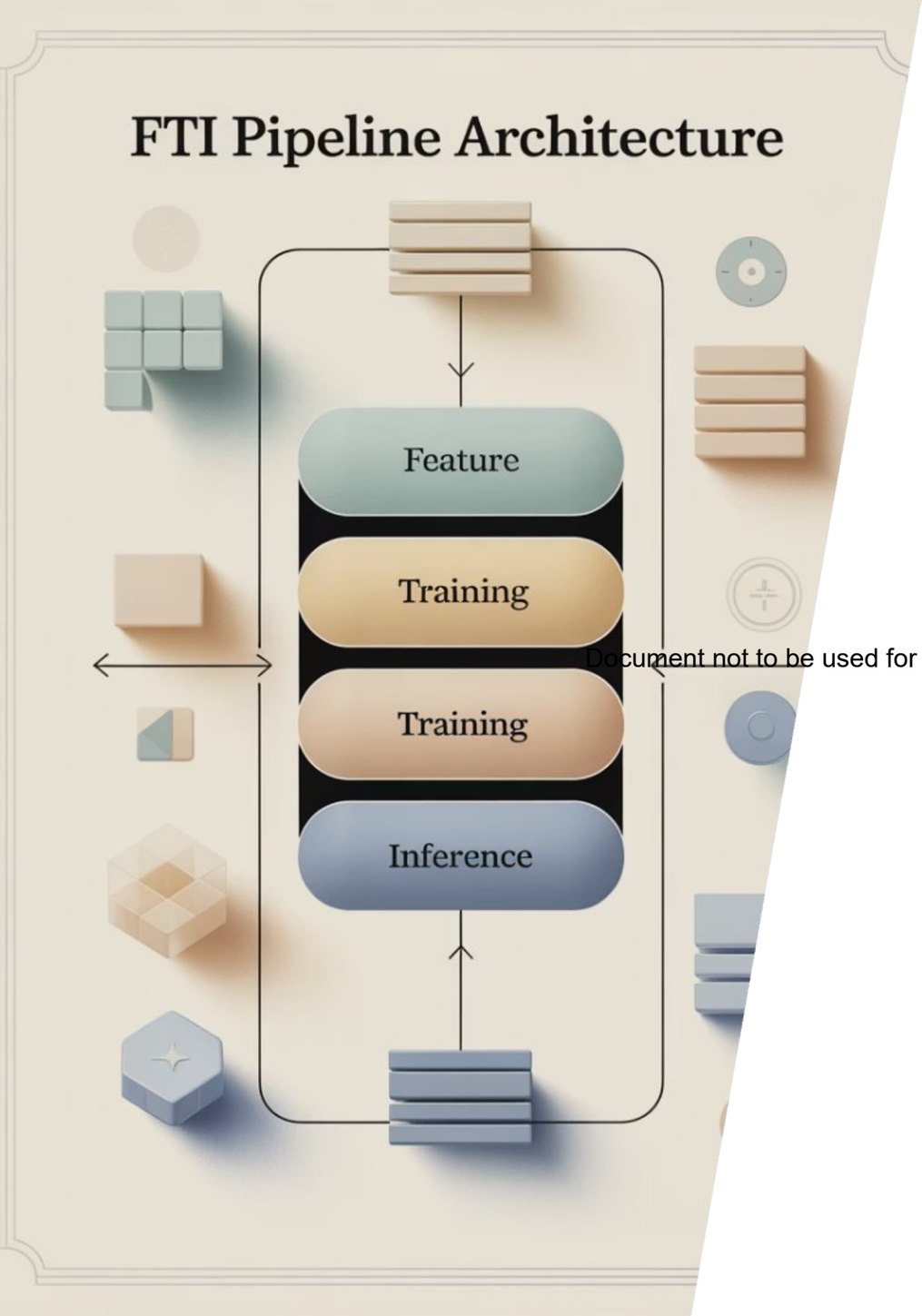
Monolith vs FTI pipelines

Monolithic approach

- Hard to debug
- Difficult to test
- Tight coupling
- Team bottlenecks

FTI approach

- Easy to debug
- Testable components
- Loose coupling
- Team autonomy



FTI Pipeline Architecture

Feature pipeline

Transforms raw data into ML-ready features and labels. Outputs to feature store.

Training pipeline

Creates models using features and labels. Outputs registered models.

Inference pipeline

Generates predictions using features and models. Outputs to end users.

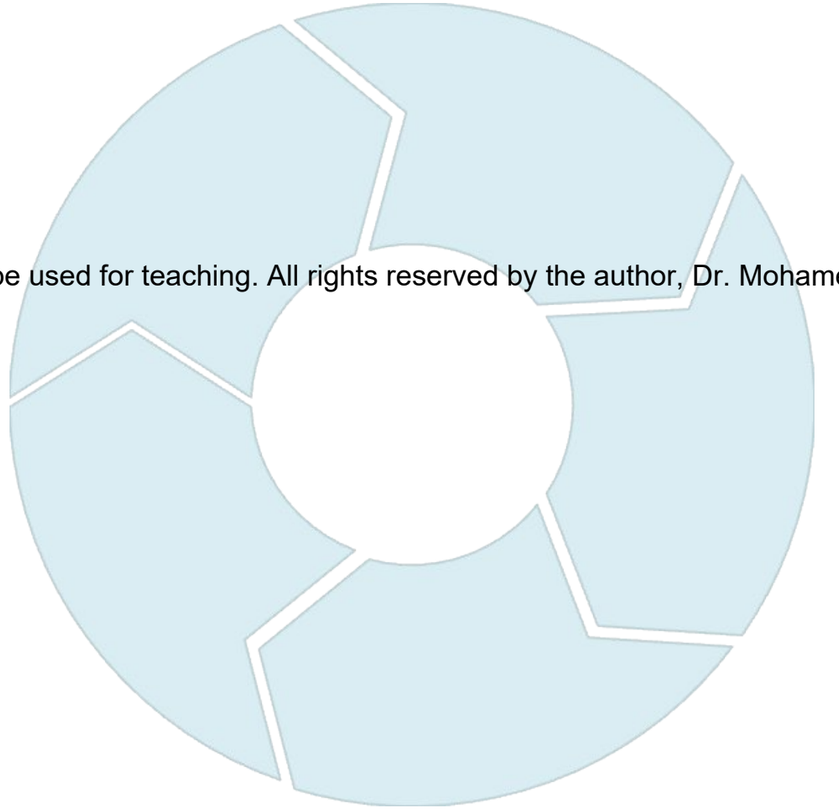
Feature pipeline

Data ingestion

Collect data from sources

Data cleaning

Handle missing values and outliers



Feature transformation

Create ML-ready features

Feature storage

Save to feature store

Feature validation

Ensure quality and consistency

Feature engineering

Domain-driven creation

- Domain expertise
- Focus on business relevance

Validation & testing

- Check distributions
- Verify transformations
- Test edge cases

Feature selection

- Remove redundant features
- Measure feature importance

Consistency

- Same logic in training and inference
- Version all transformations
- Document dependencies

Feature stores

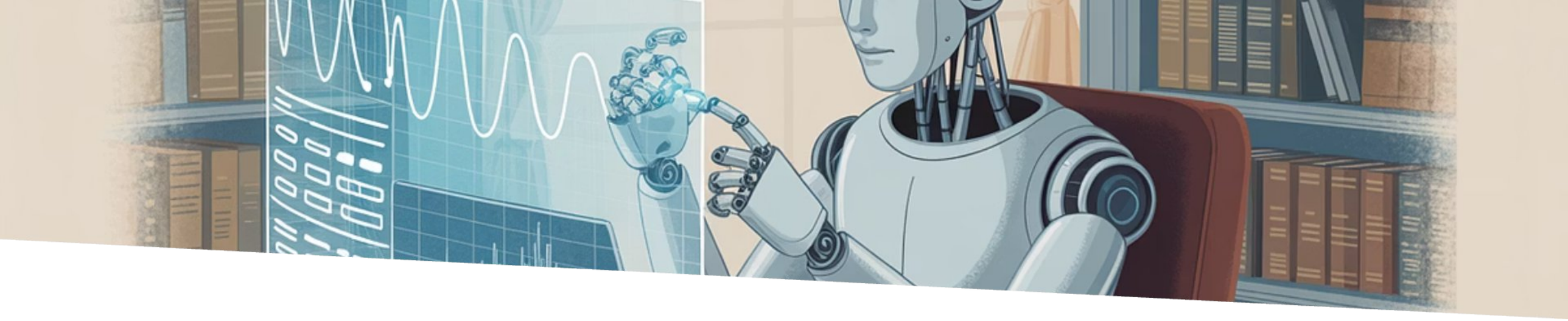
What is a feature store?

A feature store is a repository for managing and serving ML features. It connects feature engineering with model training and serving.

Think of it as a database specifically designed for ML features.

Key benefits

- Feature sharing across teams
- Consistency between training and serving
- Feature versioning and lineage tracking
- Online and offline feature serving
- Reduced redundant computations



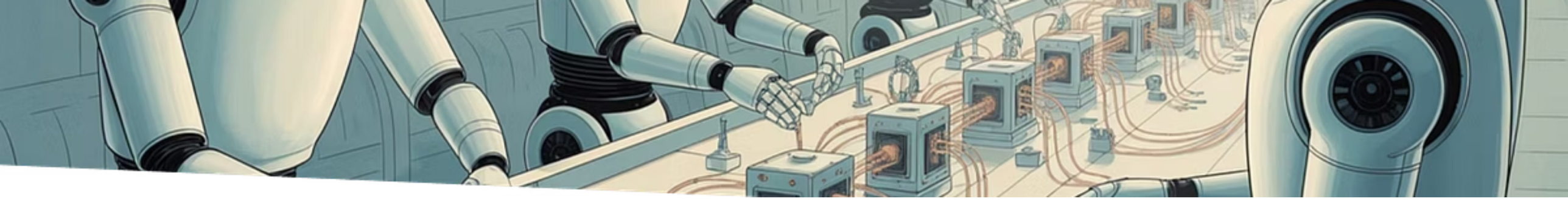
Feature pipeline debugging

Identify data quality Issues

Use profiling tools to find outliers, missing values, and distribution shifts in raw data.

Verify transformations

Check outputs after each transformation step. Visualize before and after distributions.



Proactive maintenance example (feature pipeline)

Design feature pipeline

Create a feature pipeline for predicting equipment failures. Think about what data would help predict if a machine will break down.

- Sensor data (temperature, vibration)
- Maintenance logs and history
- Environmental conditions

Define transformations

List specific feature transformations needed. Consider time-series aggregations and normalization for sensor data.

Plan testing strategy

Outline how you would validate feature quality. Define expected distributions and quality thresholds for metrics.

Feature store implementation

Choose a feature store technology. Plan how features derived from sensor data and logs will be registered and accessed.

Training pipeline

Model registration

Register validated models with metadata. Prepare for deployment.

Model validation

Evaluate model quality. Test for performance, bias, and robustness.

Model training

Train models with hyperparameter optimization. Log all experiments and metrics.

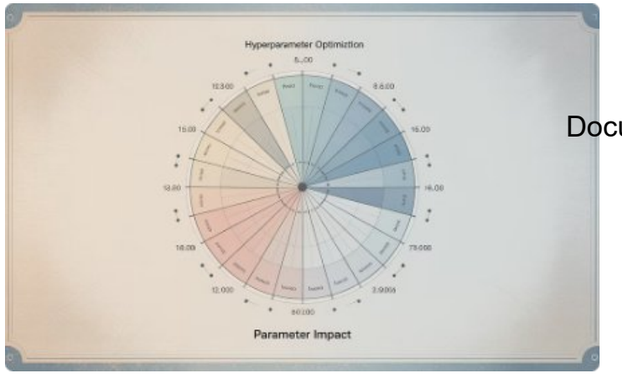
Feature retrieval

Get features from feature store. Create training datasets with proper splitting.

Model Experimentation

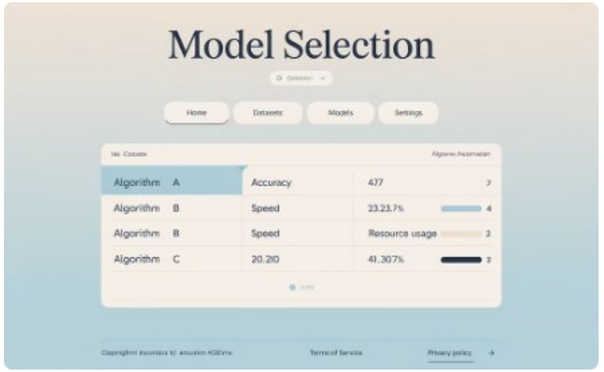
Experiment tracking

Log all model runs with parameters, metrics, and artifacts. Tools like MLflow make results searchable.



Hyperparameter tuning

Systematically search for optimal parameters. Use Bayesian optimization or grid search approaches.



Model selection

Compare models based on performance metrics. Consider tradeoffs between accuracy and inference speed.

Model validation

Performance metrics

Measure accuracy, precision, recall, and business-specific KPIs. Set clear thresholds for acceptance.

Fairness assessment

Test for bias across protected attributes. Ensure model performs equally well across all groups.

Data slicing

Evaluate performance across different segments. Identify underperforming slices that need attention.

Validation tests

Run automated test suite on models. Validate model behavior on edge cases.

Training pipeline best practices

Deterministic training

Set random seeds for reproducibility. Document all randomness sources. Use same environment for reruns.

Extensive logging

Log all parameters, metrics, and artifacts. Track resource usage and training time.

Controlled data splitting

Use stratified sampling for imbalanced data. Ensure test data resembles production data.

Configuration-driven

Drive experiments via configuration files. Avoid hardcoded parameters in code.

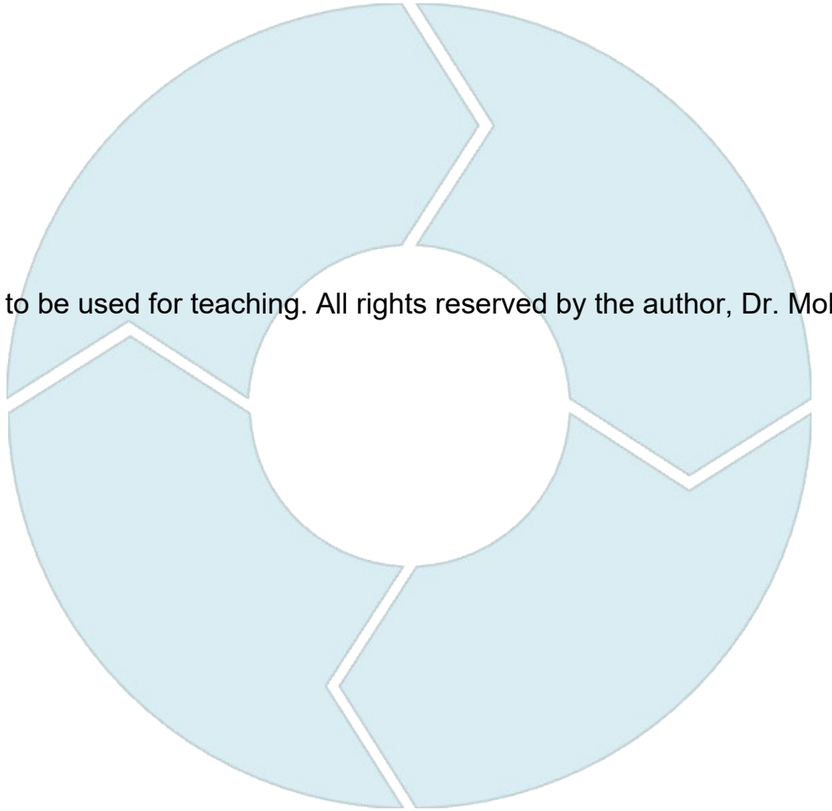
Inference pipeline overview

Feature retrieval

Get latest features from feature store

Model loading

Load appropriate model version

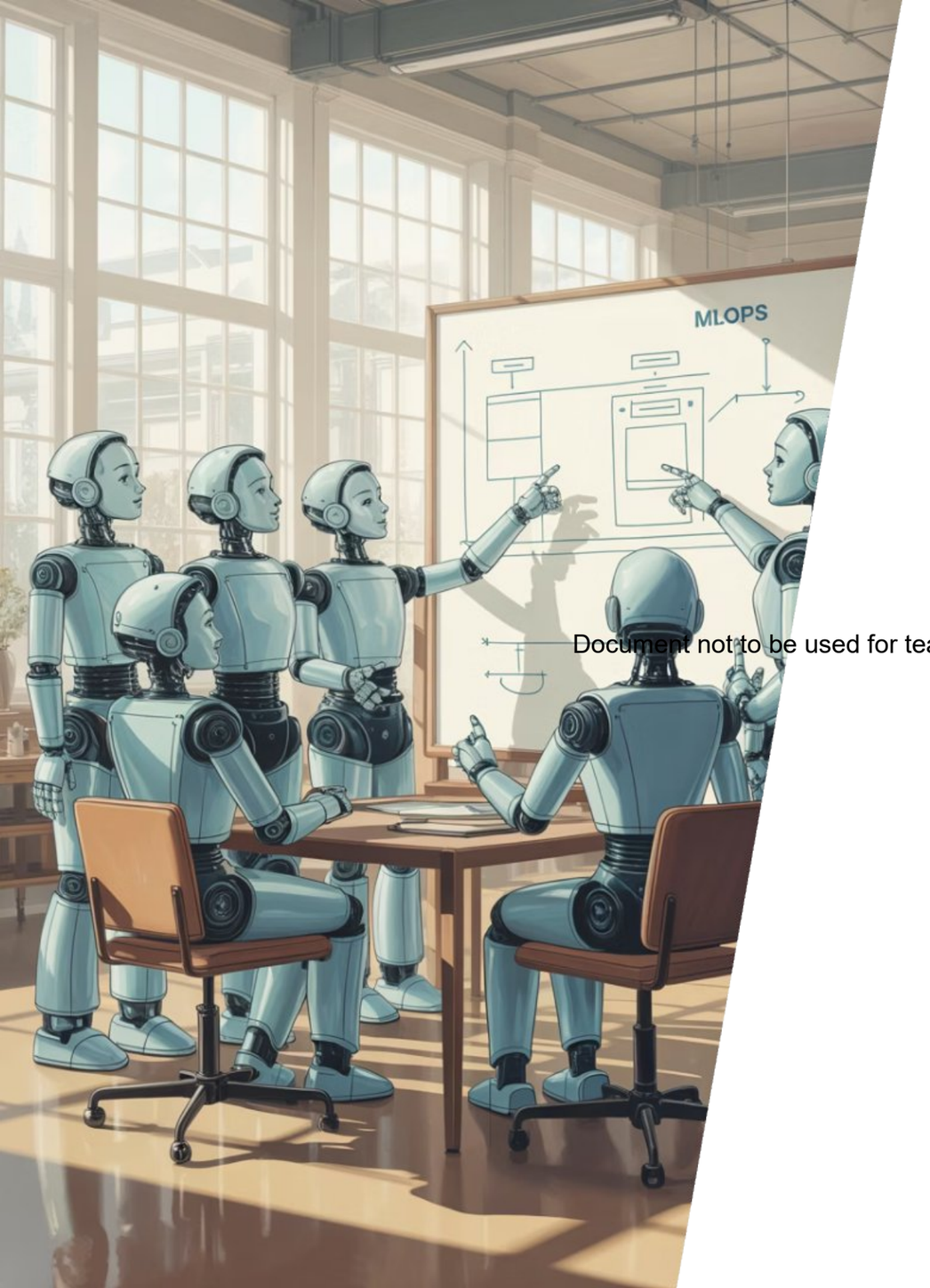


Post-processing

Format and enhance predictions

Prediction generation

Apply model to features



Practical exercice

Map your ML project

Take a current or planned ML project (e.g., phosphate production prediction). Map it to the FTI architecture. Identify components for each pipeline.

Define interfaces

Determine clear inputs and outputs for each pipeline. Document data formats and schemas.

Assign responsibilities

Outline which team members would own each pipeline. Consider required skills and knowledge.

Batch vs. Real-time Inference

Characteristic	Batch Inference	Real-time Inference
Timing	Periodic (hourly/daily)	Immediate (milliseconds)
Volume	Large data sets	Single instances
Latency	Can be high	Must be low
Infrastructure	Optimized for throughput	Optimized for latency
Use Cases	Reports, daily recommendations	Fraud detection, search results

Model serving technologies

API frameworks

- Flask: Simple Python web framework
- FastAPI: Modern, high-performance framework
- Django: Full-featured web framework

Model servers

- TensorFlow Serving: For TF models
- TorchServe: For PyTorch models
- ONNX Runtime: For framework-agnostic serving

Orchestration

- Docker: Container packaging
- Kubernetes: Container orchestration
- Vertex Ai Pipeline, Kubeflow

Inference Performance Optimization

Model optimization

- Quantization: Reduce precision
- Pruning: Remove unnecessary weights
- Distillation: Create smaller student models
- Graph optimization: Fuse operations

Serving optimization

- Batching: Process multiple requests
- Caching: Store common predictions
- Precomputing: Generate results ahead of time
- Asynchronous processing: Non-blocking APIs

Infrastructure optimization

- Horizontal scaling: Add more servers
- Vertical scaling: More powerful machines
- Hardware acceleration: GPUs, TPUs
- Deployment location: Edge vs. cloud

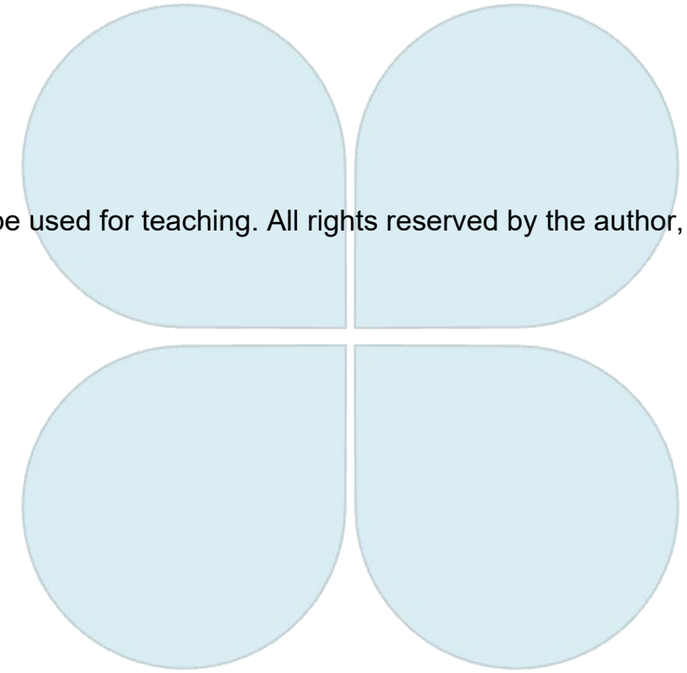
Deployment & Monitoring

Deployment

Safely releasing models to production

Monitoring

Tracking model health and performance



Governance

Ensuring compliance and documentation

Retraining

Updating models with new data

Deployment strategies

Strategy	Key Benefit	Challenge
Blue/Green	Instant rollback	Resource intensive
Canary	Limited exposure	Complex traffic management

Blue/Green Deployment

Process

- Two parallel environments
- Blue: current version
- Green: new version

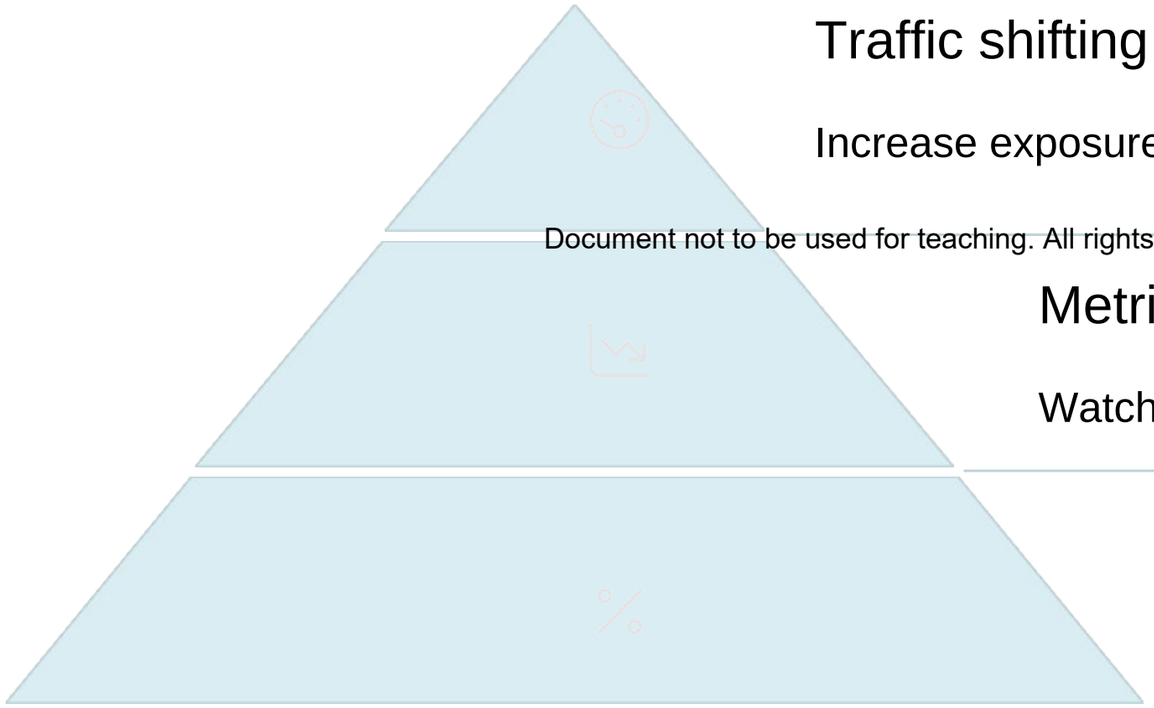
Advantages

- Zero downtime
- Quick switch
- Simple rollback

Challenges

- Double resources
- Database migrations

Canary deployment



Traffic shifting

Increase exposure on success

Metrics monitoring

Watch for issues in real traffic

Gradual rollout

Start with small traffic percentage

Continuous integration/Continuous delivery

Code changes

Developers commit code to version control. Changes trigger automated workflows.

Automated testing

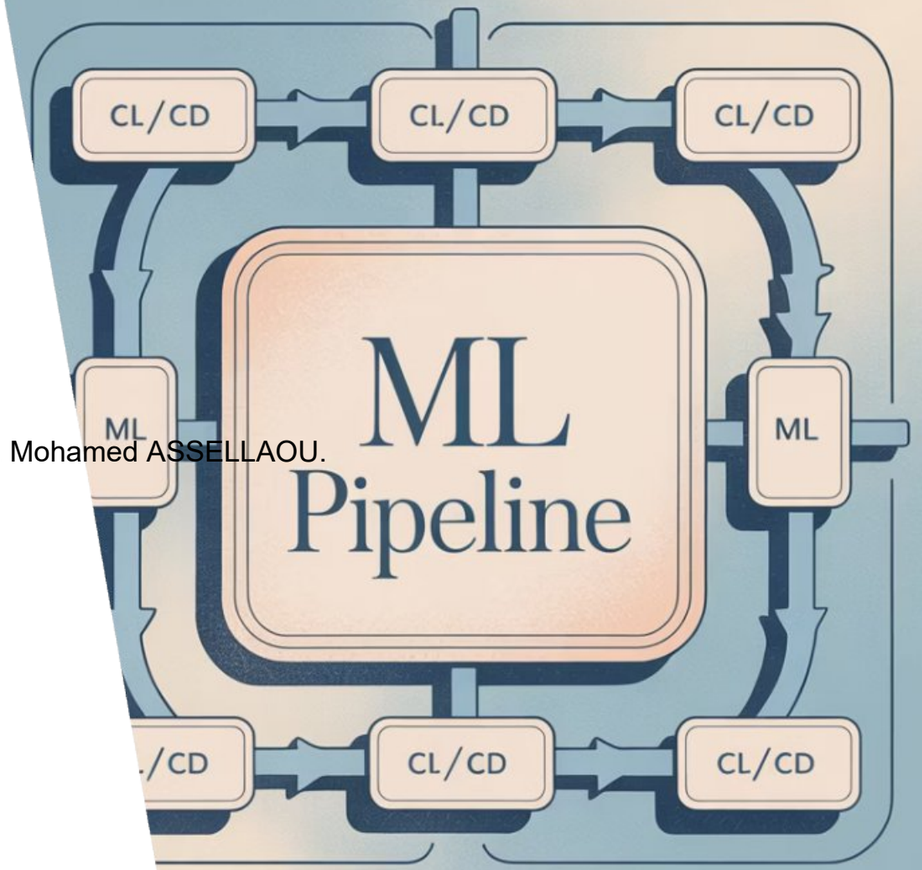
Run unit, integration, and model quality tests. Verify all components work together.

Package & release

Build deployment artifacts. Create containers or packages for each pipeline.

Deployment

Safely deploy to production using blue/green or canary strategies. Monitor rollout health.



Model monitoring

Data drift detection

Compare production data distributions with training data. Identify shifts in input patterns.

Concept drift detection

Detect changes in relationships between features and target. Often requires delayed ground truth.

Performance monitoring

Track accuracy, precision, recall, and business metrics. Alert when performance degrades.

Alerting system

Set up automated alerts for drift thresholds. Create action plans for different alert types.

MLOps observability

Three pillars of observability

Comprehensive visibility into ML systems requires three complementary approaches.

- Logs: Detailed system events
- Metrics: Quantitative performance data
- Traces: Request flows through system

ML-specific observability

Machine learning systems need additional monitoring beyond traditional software.

- Feature distributions
- Prediction distributions
- Model performance metrics
- Data quality metrics
- Explanation metrics

Conclusion

Start Small

Begin with simple pipelines. Iterate and improve continuously.

Focus on Automation

Eliminate manual steps gradually. Prioritize reproducibility in all processes and document everything.

Invest in Tooling

Choose appropriate tools for your scale. Build or buy suitable infrastructure .

Build Capabilities

Develop cross-functional MLOps skills. Remember that MLOps is about people, process, and technology.