10.15 : MLOps Deployment and Life Cycling

```
### 이번 DataCamp 강의를 수강하면서 배운점
- MLOps deployment 과정이 어떻게 이루어지는지 중점적으로 알 수 있는 강의
### 강의 내용중 궁금했던점
- 이번은 딱히 없는 것 같습니다.
### 함께 이야기 나누며 얻은 것
- PMML과 Pickle의 차이점을 설명할 때에 customizing의 자유도가 정확히 뭐
```

The modern MLOps framework

The Modern MLOps Framework and Course Overview (00:00 - 00:46)

- This course focuses on **MLOps deployment and life-cycling**, led by **Nemanja**, a Senior Machine Learning Engineer with 10 years of experience in ML tasks, now focusing on **model deployment** and **life-cycling**.
- It's a **conceptual course**, meaning it emphasizes a high-level understanding of **MLOps principles** rather than practical skills.
- **MLOps** is a combination of practices, tools, and methodologies designed to make ML workflows **automated**, **reproducible**, and **integrated** across the entire lifecycle.

Course Objectives and Big Picture of MLOps (00:46 - 01:17)

- The course introduces you to **MLOps concepts** related to the **Operations stage**—everything that happens after model training, such as deployment and monitoring.
- However, it's important to note that **MLOps** is present even in the **development phase**, preparing the model for smooth transition into production.
- In **Chapter I**, the course outlines the **value of MLOps**, explaining how it automates and integrates **model life cycle stages**, making processes faster,

reliable, and more manageable.

MLOps vs DevOps and the Risk of Technical Debt (01:17 - 02:27)

- Unlike traditional **DevOps**, **MLOps** introduces the additional complexities of handling **data** and **models** along with code.
- Organizations often start ML initiatives without fully adopting MLOps, leading to **manual workflows** and ad-hoc model monitoring.
- This approach results in **technical debt**, which is the growing cost of not automating or properly managing processes.
    - **Technical debt** occurs when quick, easy solutions are chosen over long-term, more efficient ones, leading to slower, more error-prone operations.
    - The **Google paper**, "Machine Learning: The high-interest credit card of technical debt," describes how technical debt in ML can accumulate exponentially.
        - Learn more about technical debt and read Google's paper.

ML Workflows and MLOps Maturity (02:27 - 03:15)

- **ML workflows** generally include: data collection, preparation, labeling, model selection, training, deployment, and monitoring.
- **MLOps maturity** refers to how well these workflows are automated and integrated within the larger IT infrastructure.
    - The more automated and integrated, the higher the **MLOps maturity**, leading to **faster, reproducible, and explainable** ML operations.
- **Implementing MLOps** ensures smoother transitions from model development to production, reducing errors and building trust with customers.

Focus on Operations and the Importance of Monitoring (03:15 - 03:45)

- This course emphasizes the **Operations phase** of MLOps, where the focus shifts from exploratory tasks to maintaining models in production.

- In production, **models are under scrutiny**, with their performance directly affecting users. Any unexpected behavior becomes immediately visible to customers, leaving little margin for error.
- **Monitoring** is critical to catch issues early, and MLOps ensures that systems are in place to update and fix models quickly when necessary.

Life-cycling stages

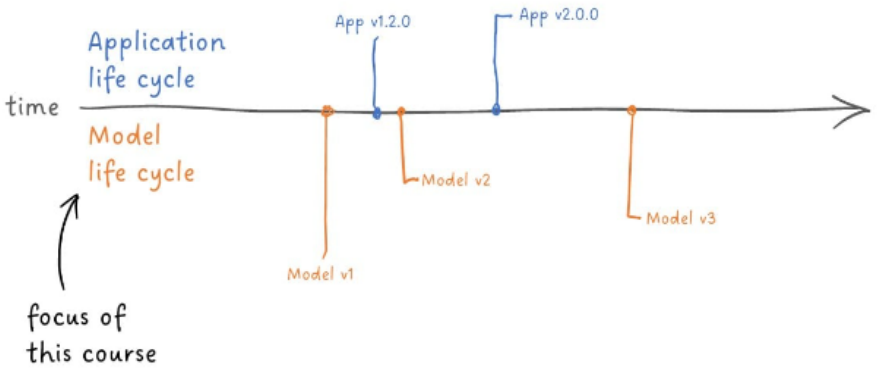
Life-Cycling Stages in Machine Learning (00:00 - 00:28)

- **Model life cycle** refers to the entire journey of a machine learning model from deployment to decommissioning. In machine learning, we also differentiate between the **ML project life cycle**, **ML application life cycle**, and **ML model life cycle**.
  - The **ML project life cycle** broadly defines the overarching effort to solve a business problem using ML.
  - If successful, the result is the development of an **ML application** and **ML models**.

ML Application vs. ML Model (00:28 - 01:35)

- **ML models** are the core predictors, such as a sales forecasting model.
- An **ML application**, however, includes the model along with other components like:
  - **Business rules** (e.g., default recommendations for users with less data),
  - **Databases** (to store features and log model outputs),
  - **GUI** (for admins to configure and troubleshoot),
  - **API** (to ensure secure and consistent communication).
- In some cases, **ML models** are tightly integrated into the application, forming a **monolithic unit**, but best practices advocate for separating the two, following a **microservice architecture**.

Application Life Cycle vs. Model Life Cycle (01:35 - 02:16)



- The **ML application** is like a car, with a long life span, while **ML models** are like tires, frequently replaced and updated as needed.
- This course focuses specifically on the **ML model life cycle**, which begins once the **trained model** is ready for deployment.

Model Deployment and Monitoring (02:16 - 02:56)

- A **trained model** and its associated resources form a **deployment package**.
- **Deployment** refers to putting the model into production, which marks the beginning of the model's life cycle.
- Once deployed, **model monitoring** ensures that the model runs smoothly and that its performance aligns with expectations.

Decommissioning and Model Archiving (02:56 - 03:39)

- Over time, a model may need to be replaced with a newer version due to better features, improved algorithms, or changes in the business environment.
- The old model is **decommissioned** and archived, a process crucial in regulated industries where **reproducibility** is required—meaning we need to be able to load and explain previous model versions even years later.

MLOps components

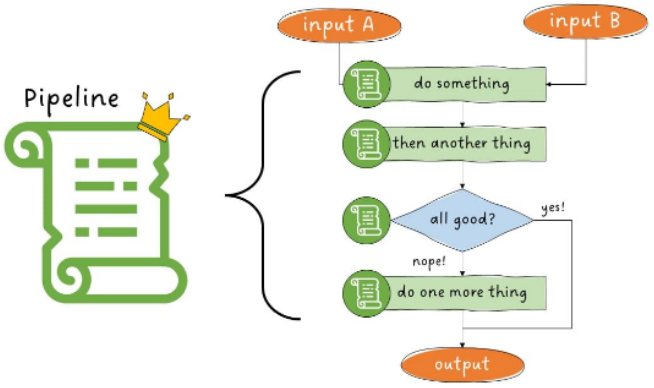
MLOps Components and Concepts (00:00 - 00:29)

- The **MLOps framework** integrates various software development concepts, such as **workflows**, **pipelines**, and **artifacts**, along with ML-specific

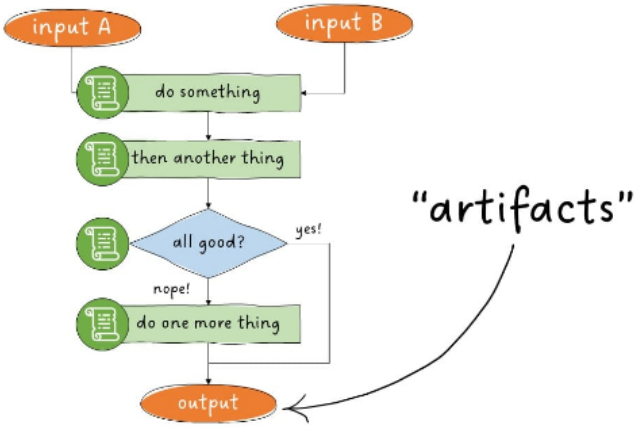
elements like the **model registry**, **feature store**, and **metadata store**.

- **MLOps** extends **DevOps** principles to include **ML workflows**, automating tasks and interactions within the ML ecosystem.

Workflows, Pipelines, and Artifacts (00:29 - 01:27)

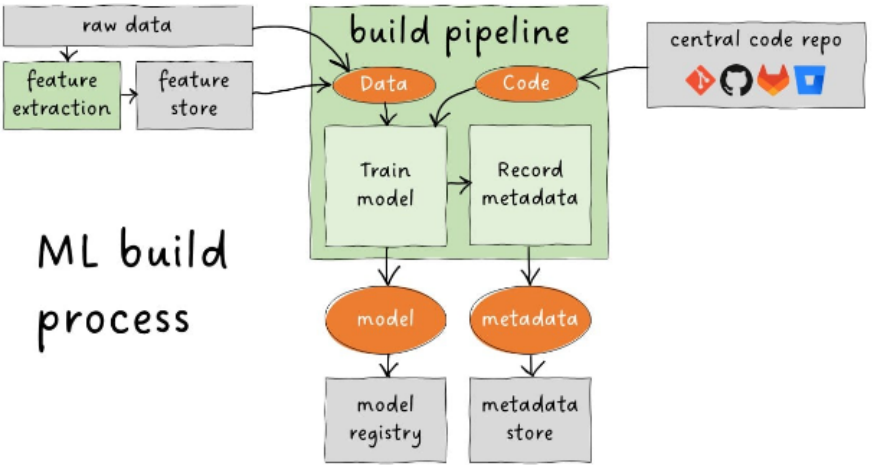


- A **workflow** is any sequence of tasks that produces outputs (artifacts) from inputs.
- These workflows can be manual, automatic, or semi-automatic. When automated, they become **pipelines**, which are essential in both IT and ML operations.



- **Artifacts** are the outputs of pipelines, ranging from data to the trained model.

ML-Specific Build Pipelines (01:27 - 02:14)

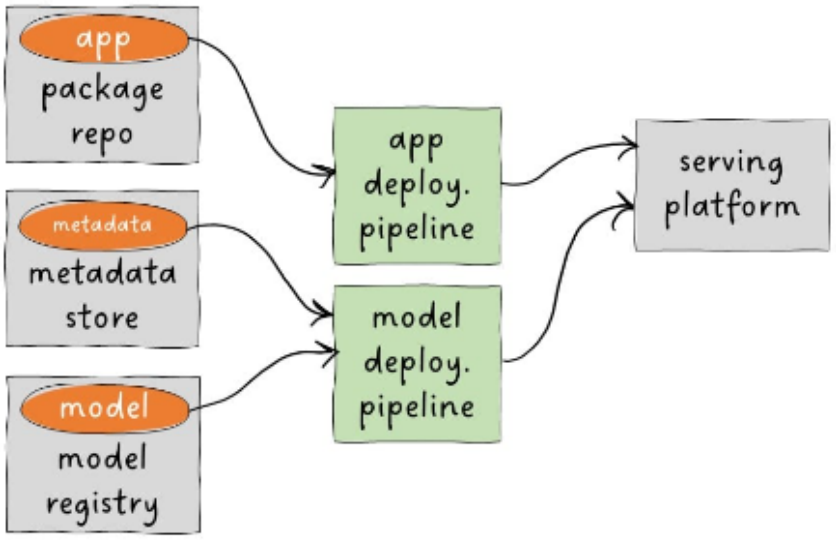


- **Build pipelines** in MLOps focus on creating deployable **ML applications** and **models**.
- In MLOps, we manage two distinct build workflows: one for the **ML model** and another for the **ML application** that serves the model.

Model Build and Feature Store (02:14 - 02:47)

- The **model build pipeline** includes training the model using code and data.
- **Processed data** used in training is stored in a **feature store**.
- After successful training, we produce the **trained model artifact** and additional metadata required for deployment.

Model Registry, Metadata Store, and Deployment Pipelines (02:47 - 03:43)



- The **model artifact** is stored in a **model registry**, which tracks and versions models.
- The **metadata** (complementary data for deployment) is stored in a **metadata store**.
- **Deployment pipelines** then place the **ML app package** and **model package** onto the target serving platform, readying the model for production.

Monitoring and MLOps Architecture Overview (03:43 - 04:01)

- Once deployed, we begin **monitoring** to ensure the model is running and performing as expected.
- This high-level overview provides a glimpse into **MLOps architecture**, which will be further explored in detail in subsequent chapters.

Deployment-driven development

Deployment-Driven Development and Key Concerns (00:00 - 00:22)

- **MLOps** doesn't just start at operations; it must be considered from the early stages of development to avoid deployment issues.
- Similar to a race car driver anticipating a curve, ML engineers need to think about deployment **before** reaching the operations stage.

Deployment Scenario (00:22 - 00:55)

- Imagine having to deploy a colleague's model. Concerns about **infrastructure compatibility**, **transparency**, **reproducibility**, **data validation**, **monitoring**, and **debugging** should immediately arise.
- The first step is to ensure that the model can run on the target infrastructure. For instance, building a model that requires significant resources but needs to run on a low-power device (e.g., a smartphone) could be a show-stopper.

Transparency and Reproducibility (00:55 - 02:12)

- It's critical to understand **who trained the model**, **when**, **with which data**, **which script**, and **which hyperparameters**.
- Every model should come from a **fully transparent pipeline** that is versioned, allowing anyone to **reproduce the exact model** at any later point.
- Logging experiments during exploration ensures that future maintainers can avoid reinventing the wheel, utilizing the **metadata store** to capture this information.

Input Data Validation (02:12 - 02:40)

- When users provide invalid input (e.g., a negative value where a positive one is expected), we need systems in place to catch these issues.
- **Data profiles** or **expectations** can help by setting clear references for valid input, and these should be stored within the **model metadata**.

Monitoring and Debugging (02:40 - 03:17)

- Monitoring the model's **performance** over time requires the app to log inputs and predictions, allowing for detection of performance deterioration.
- Effective **debugging** hinges on robust logging. Every significant event, input, and output should be logged to help trace errors and bugs. **Logging** is crucial for any debugging process, and purpose-built tools should be used to manage this effectively.

Making Code Changes (03:17 - 03:54)

- As bugs inevitably arise, we must feel confident making changes to the ML app's code. This confidence comes from having a solid collection of **unit**

tests, **integration tests**, and other forms of testing (e.g., stress tests and deployment tests) in place, ensuring that future changes don't introduce new bugs.

Conclusion (03:54 - 04:00)

- While these concepts may seem overwhelming, they will soon become second nature as you continue with MLOps. Mastering them will allow you to confidently manage models and avoid potential pitfalls.

Profiling, versioning, and feature stores

Data Profiling (00:12 - 01:13)

- **Data profiling** is the automated analysis of data to create **high-level summaries**, or **data profiles**, that validate and monitor data in production.
- These profiles help **flag incorrect inputs**, **determine when to retrain models**, and identify **data drift**. Without them, you risk incorrect predictions or failure to recognize when retraining is needed.
- **Data profiles** should be included in the **model metadata** and stored in the **metadata store**. A popular tool for profiling is **Great Expectations**.

Data Versioning (01:13 - 02:15)

- **Data versioning** ensures **reproducibility** by recording the exact version of the data used to train the model.
- Instead of storing the full dataset in the model package, we store a **pointer** to the data and a **fingerprint** to verify its integrity.
- For additional precision, we can store information about the **train-test split** used for performance estimation. A popular tool for data versioning is **DVC (Data Version Control)**.

Feature Stores (02:15 - 03:14)

- **Feature stores** are centralized databases specifically designed for storing data used in **ML training and inference**.
- They enable **cross-project feature reuse** and help reduce **feature engineering time**.

- Feature stores often include **dual databases**: one for handling large datasets for training and another optimized for retrieving records quickly during prediction.
- They also help mitigate **training-serving skew**, a common issue where models perform worse in production due to differences in training and inference data handling.

Practical Example of Skew (03:14 - 03:35)

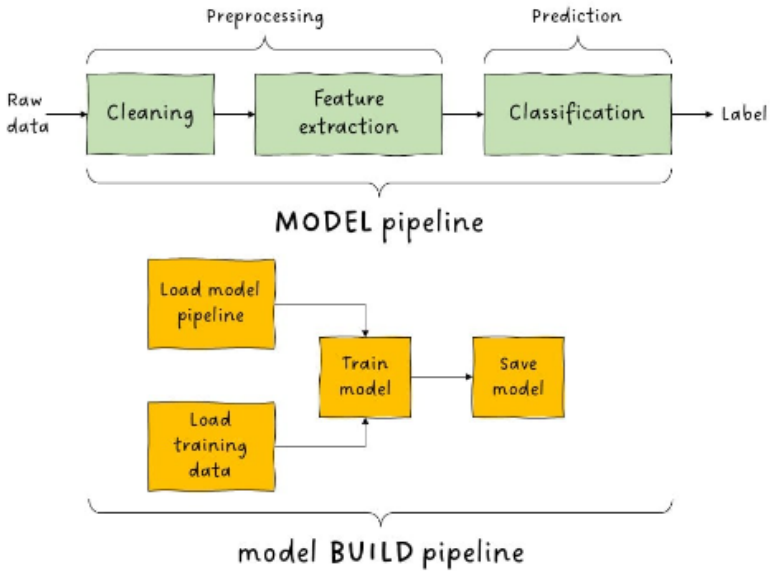
- For instance, training a spam filter on clean text emails but encountering HTML emails in production can significantly degrade model performance. **MLOps practices** prevent such mistakes by ensuring data consistency across training and production environments.

Model build pipelines in CI/CD

Model Build Pipelines in CI/CD (00:00 - 00:22)

- **Two ML-related build pipelines**: One builds the app serving the model, and the other builds (or trains) the model itself. This lesson focuses on the **model build pipeline**, the core of the MLOps framework.

Model Build Pipeline vs. Model Pipeline (00:22 - 00:53)

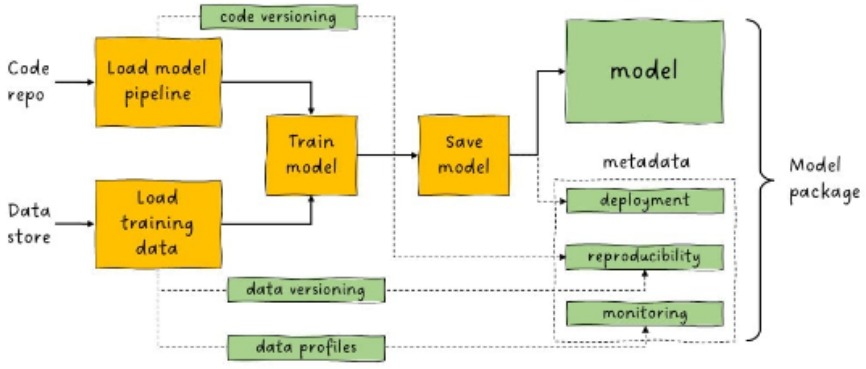


- The **model build pipeline** refers to the automated workflow that trains and saves the model, while the **model pipeline** handles the sequence of data processing steps (like cleaning and feature extraction) before a prediction is made.

MLOps-Worthy Build Pipeline (00:53 - 01:24)

- A basic model build pipeline may simply load data, train the model, and save it. However, an MLOps-compliant pipeline should also facilitate **deployment**, **reproducibility**, **monitoring**, and **CI/CD integration**.

Full Model Package and Deployment Artifacts (01:24 - 01:57)



- The build pipeline must produce a **complete model package**, including all artifacts (e.g., software dependency specifications) necessary for **deployment**. Regular **test deployments** should be performed to ensure everything works correctly.

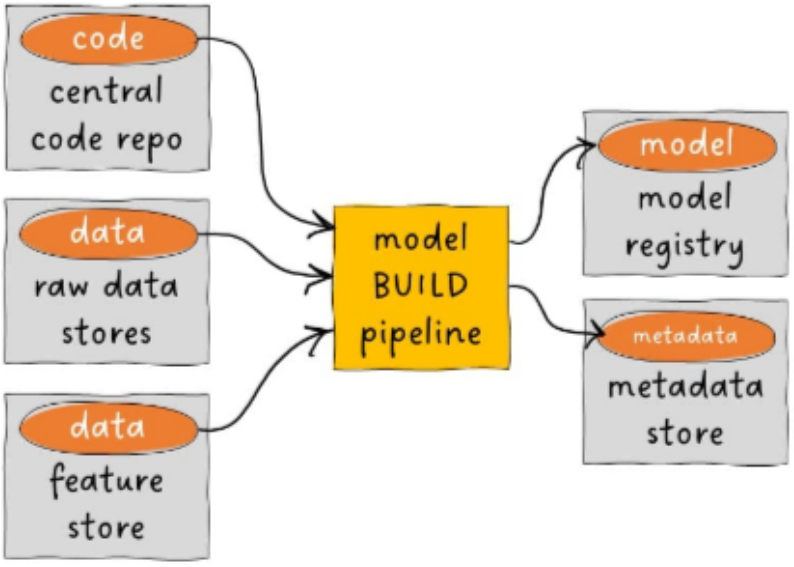
Reproducibility (01:57 - 02:37)

- **Reproducibility** is crucial for ensuring trust in the model. By versioning the **code** and **training data** and recording them in the **model metadata**, we guarantee that the model can be recreated at any point. This is essential for controlling the model production process.

Monitoring (02:37 - 02:57)

- The model build pipeline should generate **data profiles** during execution to facilitate **monitoring** and ensure the model behaves as expected in production.

CI/CD Enablement (02:57 - 03:24)



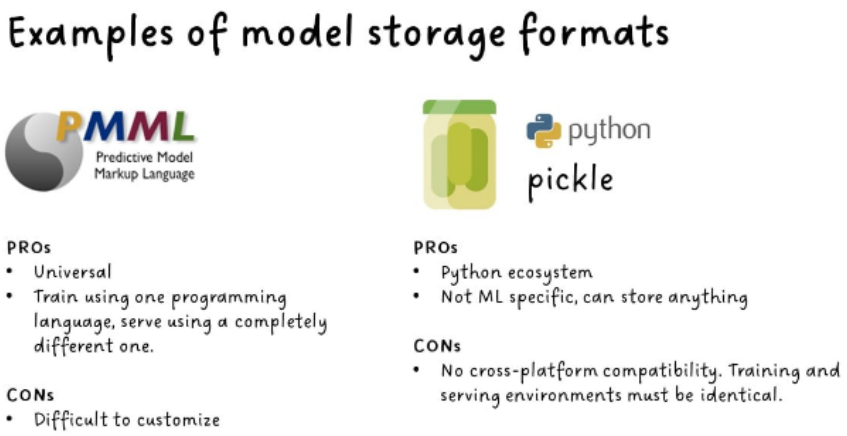
- Integrating the model build pipeline into a **CI/CD framework** ensures that models are only created with **versioned code** and **versioned data**, preventing unversioned elements from entering the process. The CI/CD platform must connect to all the relevant MLOps components.

Model packaging

Into the Wild (00:07 - 00:32)

- **Model packaging** marks the transition from **ML development** to **Operations**. To prepare for real-world deployment, our model package must ensure:
  - Smooth deployment
  - Reproducibility
  - Monitoring

Model Storage Format Options (00:32 - 01:19)



- The **trained model** is the crown jewel of the package. It can be saved in different formats depending on what the **model development framework** can produce and what the **serving framework** can load.
  - **PMML** is universal, enabling cross-language compatibility, but can be harder to customize.
  - **Pickle** (Python's common storage format) offers flexibility but lacks cross-platform compatibility.

Pickle & Dependency Management (01:19 - 02:01)

- **Pickle** can store complex models, but it limits us to Python. To mitigate compatibility issues, we need to store a list of **model dependencies** in the metadata to ensure they match on the serving side. The choice of storage format requires careful consideration.

Reproducibility (02:01 - 02:49)

- **Reproducibility** means we can **recreate the model** at any time. The key to achieving this is storing the right ingredients within the model package:
  - Pointers to the **model build pipeline** and **datasets** used during training
  - **Performance records** on the test set

Monitoring (02:49 - 03:08)

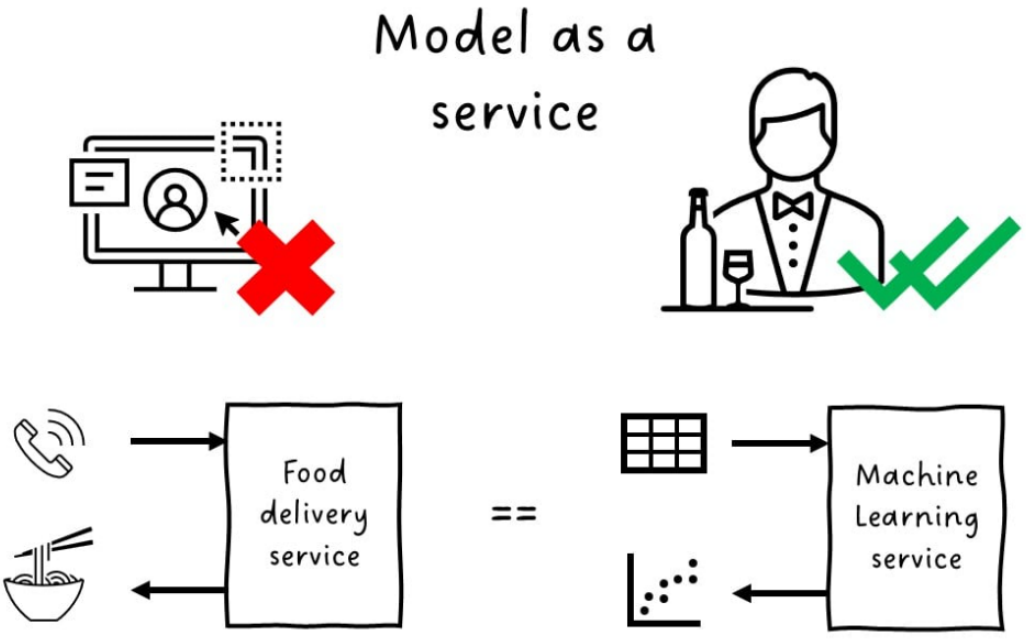
- To effectively **monitor** the model in production, we need to save **data profiles** with the package. These profiles contain **expectations** about the input and output data, ensuring consistent monitoring.

Serving modes

Serving Modes (00:00 - 00:14)

- After preparing our model package with all necessary metadata, we are now entering the **Operations phase**, where our model's life cycle truly begins.

Model as a Service (00:14 - 00:38)



- From an end-user perspective, an **ML app and model** function just like any other service. Users simply make a call to the **ML app** and expect to receive predictions. The act of providing these predictions is called **model serving**, and the way we implement it is known as the **serving mode**.

Choosing the Right Serving Mode (00:38 - 01:01)

- The first question to address when choosing a serving mode is **when** the model should generate predictions. Should the model be scheduled to run periodically, or should it respond to specific events or user requests?

Batch Prediction (01:01 - 01:42)

- **Batch prediction** (also known as **offline** or **static prediction**) is when predictions are generated on a scheduled basis, often on large datasets

(batches). It is the simplest form of serving and is ideal for use cases like monthly sales forecasts.

On-Demand Prediction (01:42 - 02:04)

- **On-demand prediction** (also known as **online** or **dynamic prediction**) generates predictions when triggered by specific events or user requests. In this mode, the response time (latency) becomes a crucial factor.

Latency and Its Importance (02:04 - 02:19)

- **Latency** is the time between the user's request and the service's response. Depending on the use case, users may tolerate varying degrees of latency, from minutes to needing predictions in real-time.

Near-Real-Time Prediction (Stream Processing) (02:19 - 02:35)

- When predictions need to be generated within several minutes, we use **near-real-time prediction**, also called **stream processing**. This is common for scenarios where immediate response isn't critical but still requires timely updates.

Real-Time Prediction (02:35 - 02:53)

- **Real-time prediction** is required when predictions must be generated in less than a second, as in cases like **credit card fraud detection**, where delayed predictions render the results ineffective.

Latency vs. Model Strength (Edge Deployment) (02:53 - 03:24)

- In some cases, **low latency** is so important that we prioritize speed over model strength. This may involve using a simpler, faster model or even deploying the model directly onto the user's device. This is called **edge deployment**, and examples include smartphones using ML for facial recognition, navigation, and image filters.

This format keeps the details while structuring the content clearly by consolidating related concepts. Let me know if you'd like to proceed with this approach!

Building the api

Building the API (00:00 - 00:57)

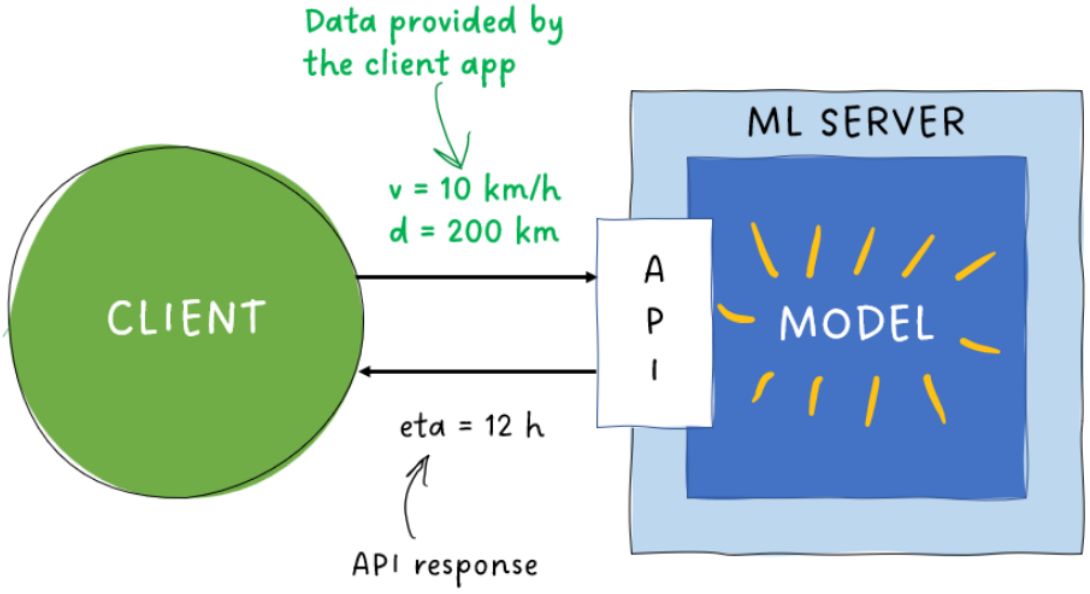
- To make our model accessible to the outside world, we need to expose it via an **API (Application Programming Interface)**. While humans use graphical user interfaces to interact with software, applications communicate through APIs. The **server** hosts the ML model, while the **client** uses it to request predictions or access remote databases.

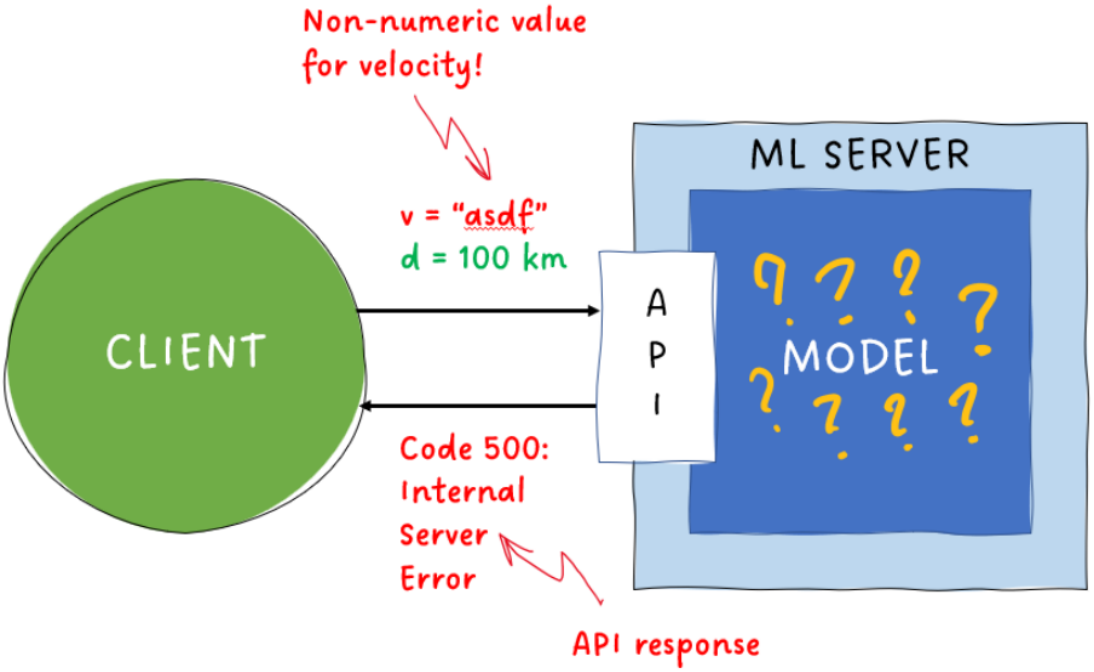
API Architectures (00:57 - 01:15)

- APIs can follow different architectural styles like **REST, RPC, or SOAP**, but for now, we'll focus on the fundamental components needed for any API, regardless of architecture.

API Example: Navigation App (01:15 - 01:41)

- In a simple API for a **Navigation app**, the client provides input like velocity and distance, which the API processes through the model to return the estimated time of arrival.





Input Validation (01:41 - 02:28)

- Input validation ensures the API handles client requests correctly. If a user sends invalid data (e.g., non-numeric velocity), the server may fail, so we validate inputs (e.g., check if velocity and distance are positive real numbers) and return clear error messages if needed. This is documented in the **request model**.

Output Validation (02:28 - 03:08)

- Output validation prevents the API from returning faulty predictions, like negative time estimates, which could cause client-side issues. By validating output values against a **response model**, we ensure only correct results are returned to the client, safeguarding our reputation.

Authentication and Throttling (03:08 - 03:40)

- To secure our API, we implement **authentication**, limiting access to specific clients. Additionally, we control the number of requests clients can make in a given period using **API throttling**.

FastAPI (03:40 - 03:57)

- open-source API framework for Python
- all essential features out of the box
- build and launch REST APIs in no time

[1] https://fastapi.tiangolo.com

- **FastAPI** is an excellent Python-based framework for building APIs. It provides essential features like input validation, output validation, authentication, and more, allowing quick and efficient API development.

Deployment progression and testing

Deployment progression and testing (00:00 - 00:59)

- After building our web API, we should **thoroughly test** it before making it available to users. In this lesson, we'll cover **unit tests, integration tests, smoke tests, load tests, and acceptance tests**, as well as different environments like **development, test, staging, and production**. These tests focus on the overall ML application rather than the model's predictive performance, examining elements like **database communication, user authentication, and logging**.

Unit Testing (01:09 - 01:43)

- **Unit tests** check if individual parts of the code (like functions) behave as expected. They should be **simple and fast**, run in a stable **TEST environment**, ensuring that development resources stay available for actual coding.

Integration Testing and Staging (01:48 - 02:42)

- **Integration tests** check how components interact and if the app can communicate with external services like databases. These tests are run in a **staging environment**, which should mimic the **production environment** as closely as possible, using the same database versions but with a **representative subset** of production data.

Smoke, Load, and Stress Tests (02:42 - 03:10)

- **Smoke tests** ensure the app can start without crashing. If successful, we move on to **load tests**, which evaluate how well the app performs under normal user load. **Stress tests** push the load to the extreme to see how the system handles overload.

User Acceptance Testing (UAT) (03:10 - 03:29)

- Once confident in the app's stability, a group of **end-users** conducts **UAT** to verify if the app meets expectations. Successful UAT grants approval for production deployment.

Caution and Prioritization (03:29 - 04:06)

- Even though extensive testing can be overwhelming, especially for Data Scientists, it's crucial to **avoid introducing faulty components** that could crash the entire system. Prioritize testing **critical parts** of the application to ensure a smooth production rollout.

Model deployment strategies

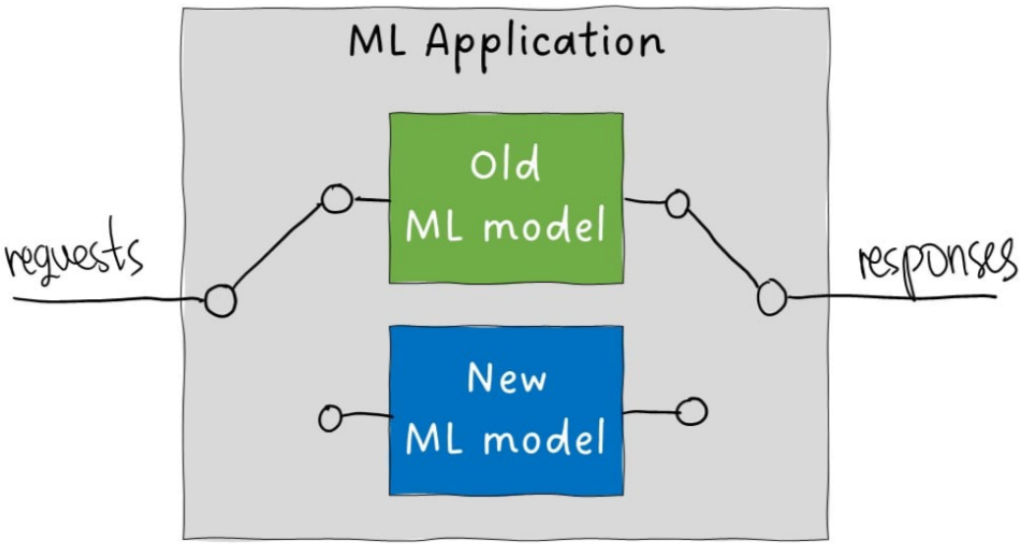
Model deployment strategies (00:00 - 00:33)

- After successfully launching the model in production, everything is running smoothly. However, with new features improving model performance, a new model package is ready for deployment. But how do we manage deploying it smoothly?

Simple swap vs. offline deployment (00:33 - 00:50)

- If we run predictions in **batch mode**, we can swap the old model with minimal risk between runs. But for **real-time services** with thousands of predictions per minute, downtime becomes costly.

Blue/Green deployment (01:01 - 01:48)



- In **blue/green deployment**, the ML app can load the new model while still using the old one. At the press of a button, we can switch all traffic to the new model. While simple, this method carries the risk of exposing all users to potential issues with the new model. However, we can quickly **roll back** to the old model if needed.

Canary deployment (01:59 - 02:32)

- A safer alternative is **canary deployment**, where we gradually send a small percentage of requests to the new model. If everything works well, **the percentage is increased** until all traffic is directed to the new model. This approach reduces risk by allowing a phased rollout.

Shadow deployment (02:32 - 03:09)

- The **safest strategy** is **shadow deployment**, where both models run in parallel, but only the old model's responses are returned to users. **The new model's outputs are saved for validation**. Though safer, running two models can **affect performance**, so it can be limited to a percentage of requests or run during off-peak hours in batch mode.

Monitoring ML services

Monitoring ML services (00:00 - 00:33)

With the ML service now live and delivering thousands of predictions, maintaining quality becomes crucial, especially as customers expect consistent service. Quality control begins with **monitoring** the system's health and ensuring acceptable latency and proper prediction handling.

Performance indicators (00:33 - 01:02)

Key performance indicators include uptime, request handling, and latency. However, the core concern is the **predictive quality** of the model, as models tend to deteriorate over time due to changes in the real-world relationship between input and output features.

Concept drift (01:02 - 01:56)

Concept drift refers to a significant change in the relationship between inputs and outputs. As reality changes, the model’s decision boundary becomes outdated, resulting in more incorrect predictions, even though the model itself hasn't changed.

Detecting concept drift (02:06 - 02:34)

Theoretical detection involves comparing predictions with the **ground truth**, but in practice, obtaining ground truth is often delayed (verification latency) or expensive. In some cases, such as financial decisions, it may never be available.

Alternative monitoring strategies (03:10 - 03:49)

Input feature monitoring involves checking for shifts in the relationship between input features, also known as **covariate shift**. While it is a good indicator of concept drift, it is not foolproof, as either phenomenon can occur independently or simultaneously.

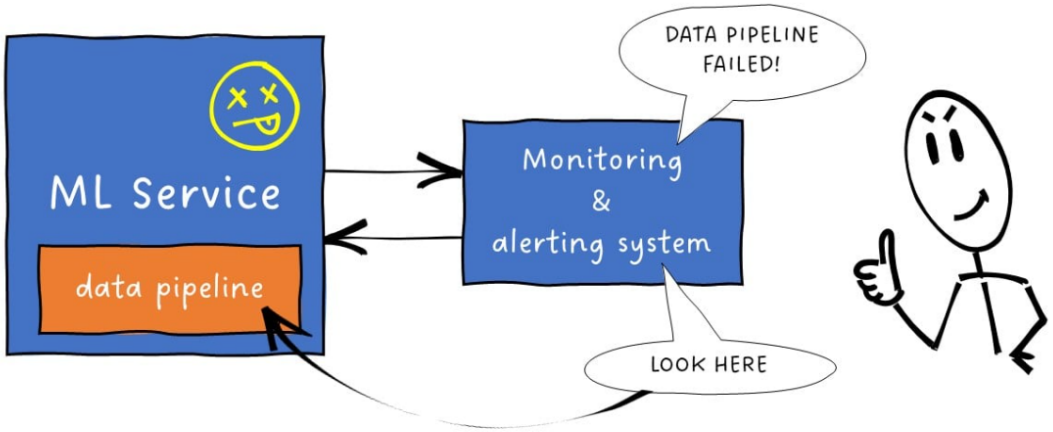
Output monitoring (03:49 - 04:10)

To supplement input monitoring, **output monitoring** focuses on changes in the distribution of model outputs. Although ground truth is the best measure, shifts in output distribution can also signal that the model requires maintenance due to reality changes.

Monitoring and alerting

Complex ML Services and Failure Points (00:34 - 01:04)

Our ML service consists of many components, such as the model, data pipeline, API, and hardware, each of which can fail. A good monitoring system not only detects that something is wrong but also helps pinpoint where the issue lies.



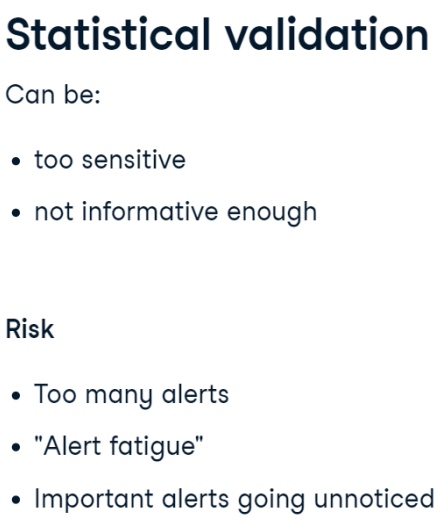
Granular Logging and Data Pipeline Monitoring (01:04 - 01:53)

Detailed logging throughout the service is essential. For example, if 5% of requests have high latency, we need to know which users were affected and how much data was sent. Similarly, we must monitor and validate each component of the data pipeline individually to catch issues early.

Data Profiles and Validation (01:53 - 02:37)

Data profiles, or data expectations, help validate the inputs and outputs of the service by setting acceptable value ranges and relationships between features. These profiles ensure that new data is properly monitored against the expected statistical distributions and constraints.

Statistical Validation and Alert Fatigue (02:37 - 03:09)



Statistical validation is sensitive to minor changes, which may generate too many alerts, leading to **alert fatigue**, where critical issues may be ignored. Therefore, it's crucial to select monitoring metrics and alert thresholds carefully to avoid this problem.

Alerting and Incident Handling (03:09 - 03:50)

When an issue arises, alerts should be sent promptly to the appropriate personnel. After resolving the incident, it's important to record the root cause and resolution steps. Google's analysis of a decade of ML pipeline incidents showed that most issues were not ML-related, providing valuable insight into the importance of monitoring non-ML components.

Centralized Monitoring Service (03:50 - 04:09)

For maximum reliability, a centralized monitoring service should be used to track all ML services in one place, ensuring a high standard of service quality for users.

Model maintenance

Model Maintenance: Introduction (00:00 - 00:20)

In this video, we focus on maintaining ML models by addressing any deterioration in performance. While we previously discussed catching system anomalies, now we shift our attention to maintaining and improving our ML model.

Model Deterioration and Solutions (00:20 - 00:40)

When a model's performance falls below acceptable levels, we have two main choices: improve the model itself (model-centric approach) or improve the quality of the training data (data-centric approach).

Model-Centric Approach (00:40 - 01:05)

The model-centric approach focuses on refining the model, experimenting with different algorithms, or deriving better features from fixed datasets. This is common in ML competitions, where datasets are pre-set.

Data-Centric Approach (01:05 - 01:37)

In real-world scenarios, we have more flexibility to clean and enrich datasets. The data-centric approach has gained popularity as it often provides better performance gains compared to focusing solely on model experimentation.

Improving Data Quality (01:37 - 02:18)

The focus of data-centric development is improving the quality of the dataset. More relevant features and better labels (closer to ground truth) lead to improved models. Poor labels, such as those collected from inaccurate sources, inevitably lead to poor model performance.

Labeling Tools and Efficiency (02:18 - 02:55)

Labeling data can be error-prone, but specialized labeling tools can improve both efficiency and accuracy by providing interfaces, suggesting impactful data points for labeling, and detecting mistakes.

Human-in-the-Loop Systems (HIL) (02:55 - 03:43)

In Human-in-the-Loop systems, humans and ML models collaborate. For instance, an ML model for medical diagnostics makes predictions, but if its confidence is low, a doctor makes the final decision. Each human intervention provides additional labeled data to improve the model.

Rebuilding and Testing the Model (03:43 - 04:32)

After constructing a new training dataset, we rerun the model build pipeline. If the new model outperforms the old one, we deploy it. Otherwise, we continue experimenting, utilizing tools like MLFlow Tracking to document experiments

and avoid redundant tests. MLOps practices ensure that model maintenance is fast and efficient.

Model Governance: Introduction (00:00 - 00:08)

In this final lesson, we explore the crucial concept of Model Governance and its importance in managing machine learning models effectively.

The Growing Impact of ML (00:08 - 01:02)

Machine learning models are used in various industries to make complex, high-stakes decisions. In scenarios like banking, a poorly performing model could result in catastrophic outcomes, such as a bank's bankruptcy, which could have ripple effects across economies. Therefore, it's critical to ensure careful oversight of model deployment to prevent significant risks and costs.

Defining Model Governance (01:02 - 01:25)

Model governance refers to how an organization controls access, implements policies, and tracks the activities and outcomes of its models. Effective governance minimizes financial and reputational risks.

Governance Across the ML Lifecycle (01:25 - 02:42)

Governance applies to all phases of an ML project. During the design phase, organizations must consider ethical concerns, privacy issues, and bias detection. During development, documentation, versioning, and data quality assurance must be defined. Before going live, controls such as API security and monitoring need to be established, and audit trails should be maintained.

Industry-Specific Governance (02:42 - 03:34)

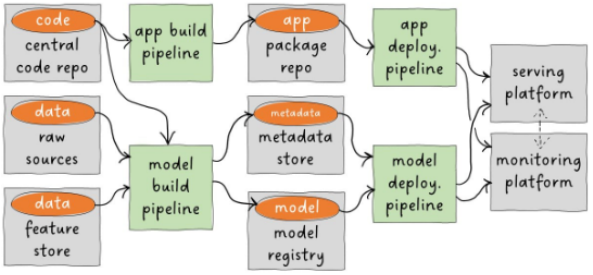
In some industries, organizations have flexibility in setting governance protocols, while in others, such as finance, strict local and international regulations dictate governance. Models with higher risk, like fraud detection, undergo more stringent scrutiny than less critical models like recommendation systems. The risk level is determined by both the impact and frequency of the model's decisions.

Governance Importance and Conclusion (03:34 - 04:18)

Although governance may introduce some friction in ML projects, it is essential to avoid reckless practices that can cause harm. A well-established

governance framework will ultimately ensure that ML models deliver business value and operate safely, reducing risks as the MLOps framework matures.

Principles above all



#automation

#collaboration

#efficiency

#transparency

#user-satisfaction

mlflow great_expectations nannyML DVC