MLOPS

ZZAK00

February 8, 2022

Abstract

Grace Hopper -: In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

I. MLOPS

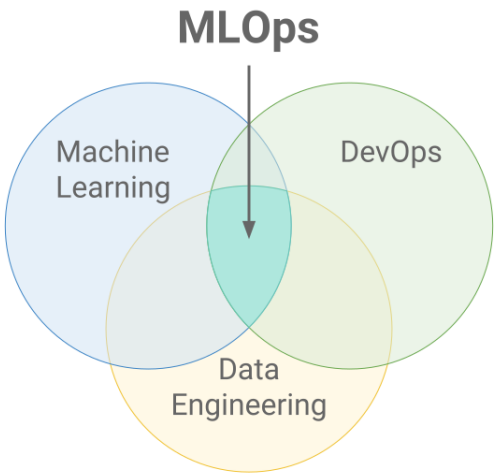


Figure 1: *ML Ops.*

MLOps a set of practices that aims to deploy and maintain machine learning models in production reliably and efficiently. The word is a compound of *"machine learning"* and the continuous development practice of DevOps in the software field. Machine learning models are tested and developed in isolated experimental systems. When an algorithm is ready to be launched, MLOps is practiced between Data Scientists, DevOps, and Machine Learning engineers to transition the algorithm to production systems. Similar to *DevOps* or *DataOps* approaches, MLOps seeks to increase automation and improve the quality of production models, while also focusing on business and regulatory requirements. While MLOps started as a set of best practices, it is slowly evolving into an independent approach to ML lifecycle management. MLOps applies to the entire lifecycle - from integrating with model generation *(software development lifecycle, continuous integration/continuous delivery)*, orchestration, and deployment, to health, diagnostics, governance, and business metrics. According to **Gartner**, MLOps is a subset of *ModelOps*. MLOps is focused on the operationalization of ML models, while ModelOps covers the operationalization of all types of AI models

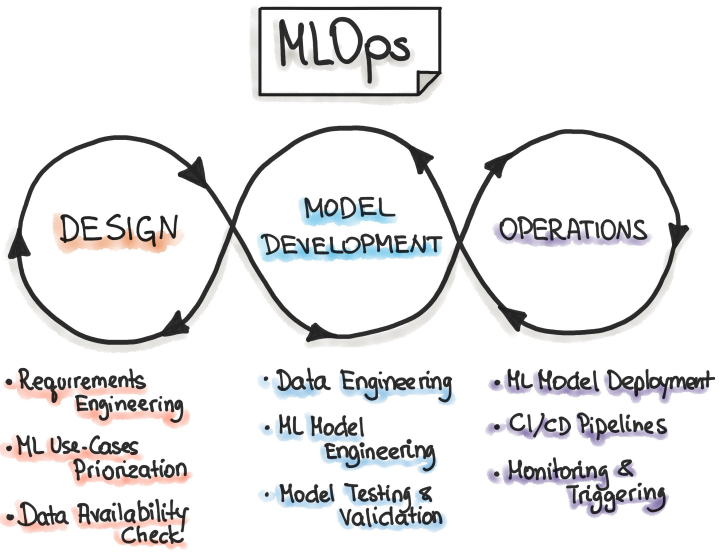


Figure 2: *ML OPS.*

II. Need to know !

i. AGILE

Agile is a time-bound, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver all at once.

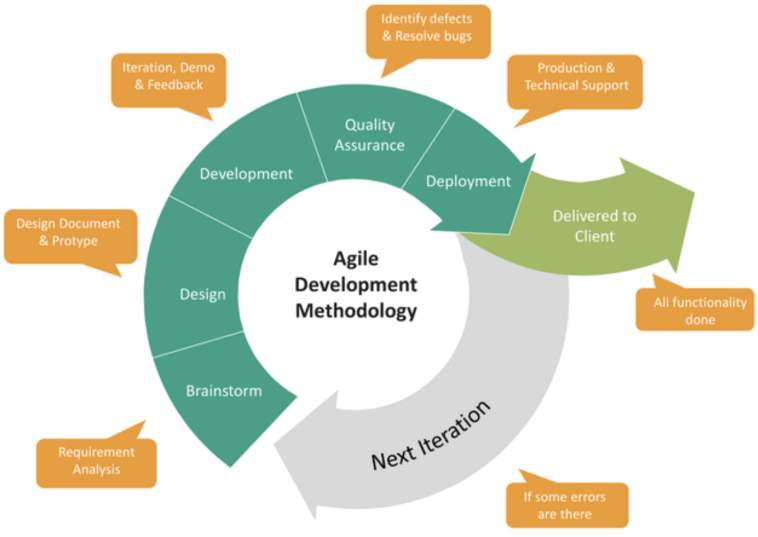


Figure 3: *Agile Software Development .*

ii. DevOps

DevOps is an approach to software development that accelerates the build lifecycle (formerly known as release engineering) using automation. DevOps focuses on continuous deployment of software by leveraging on-demand IT resources and by automating integration, test, and deployment of code. This merging of software development (“dev”) and IT operations (“ops”) reduces time to deployment, decreases time to market, minimizes defects, and shortens the time required to resolve issues.

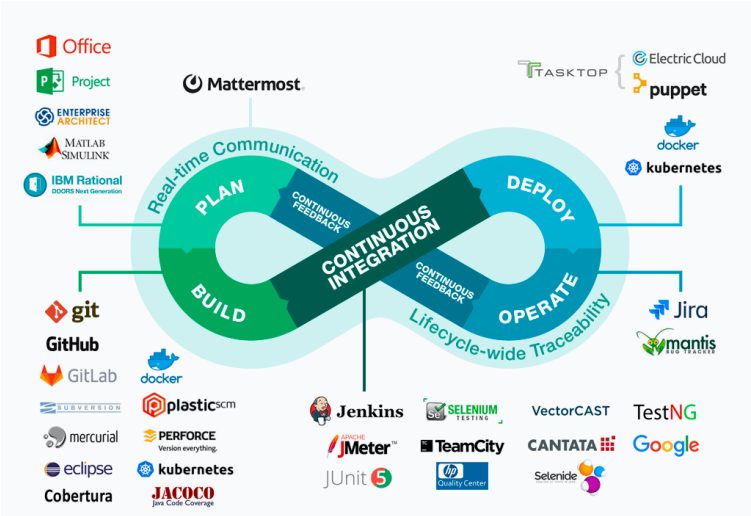


Figure 4: *Data Ops.*

iii. Dev Ops

DataOps is a collection of technical practices, workflows, cultural norms, and architectural patterns that enable:

- Rapid innovation and experimentation delivering new insights to customers with increasing velocity.
- Extremely high data quality and very low error rates.
- Collaboration across complex arrays of people, technology, and environments.
- Clear measurement, monitoring, and transparency of results.

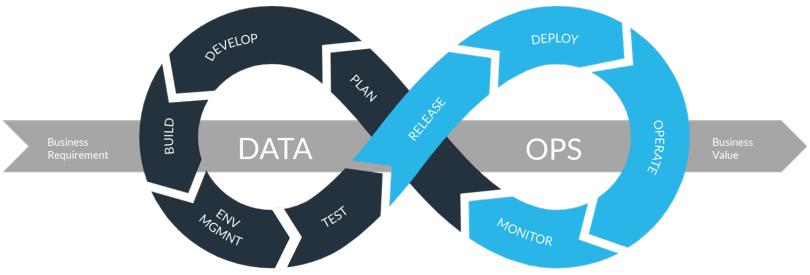


Figure 5: *Data Ops.*

III. ML OPS WORKFLOW

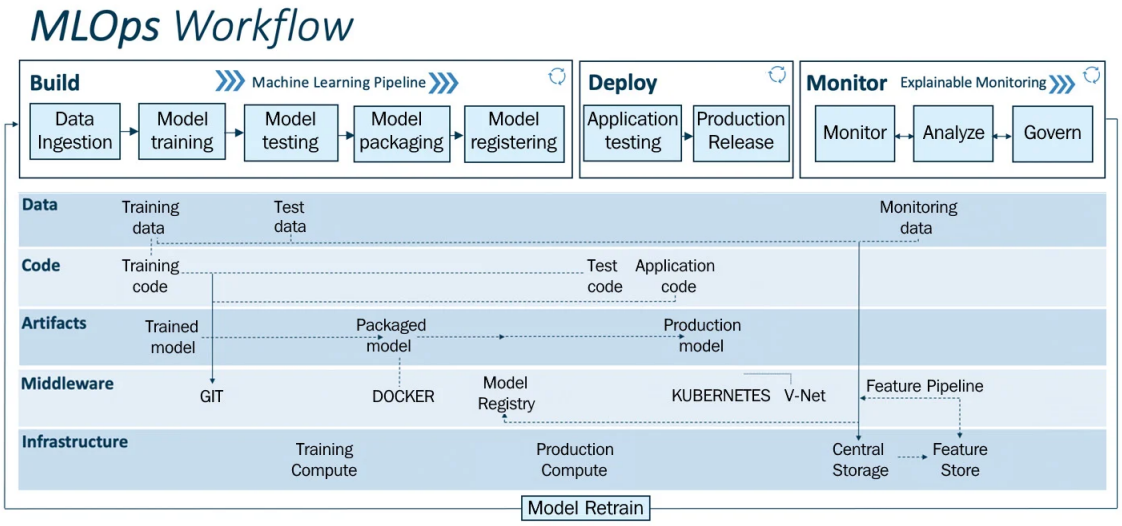


Figure 6: *MLOps workflow.*

i. Use Case:

- Data: The pet park has given you access to their data lake containing 100,000 labeled images of cats and dogs, which we will use for training the model.
- Infrastructure: Public cloud (IaaS).

ii. The Machine Learning Pipeline

The build module has the core ML pipeline, and this is purely for training, packaging, and versioning the ML models. It is powered by the required compute (for example, the CPU or GPU on the cloud or distributed computing) resources to run the ML training and pipeline.

- **Data ingestion**: This step is a trigger step for the ML pipeline. It deals with the volume, velocity, veracity, and variety of data by extracting data from various data sources (for example, databases, data warehouses, or data lakes) and ingesting the required data for the model training step. Robust data pipelines connected to multiple data sources enable it to perform extract, transform, and load (ETL) operations to provide the necessary data for ML training purposes. In this step, we can split and version data for model training in the required format (for example, the training or test set). As a result of this step, any experiment (that is, model training) can be audited and is back-traceable.
  Use Case Implementing:
  As you have access to the pet park's data lake, you can now procure data to get started. Using data pipelines (part of the data ingestion step), you do the following:
  - Extract, transform, and load 100,000 images of cats and dogs.
  - Split and version this data into a train and test split (with an 80% and 20% split).Versioning this data will enable end-to-end traceability for trained models.

Congrats – now you are ready to start training and testing the ML model using this data.

- **Model training**: After procuring the required data for ML model training in the previous step, this step will enable model training; it has modular scripts or code that perform all the traditional steps in ML, such as data preprocessing, feature engineering, and feature scaling before training or retraining any model. Following this, the ML model is trained while performing hyperparameter tuning to fit the model to the dataset (training set). This step can be done manually, but efficient and automatic solutions such as Grid Search or Random Search exist. As a result, all important steps of ML model training are executed with a ML model as the output of this step.
  Use Case Implementing:
  Train and fine tune a CNN classification Model.

- **Model testing**: In this step, we evaluate the trained model performance on a separated set of data points named test data (which was split and versioned in the data ingestion step). The inference of the trained model is evaluated according to selected metrics as per the use case. The output of this step is a report on the trained model's performance
  Use case implementation: We test the trained model on test data (we split data earlier in the Data ingestion step) to evaluate the trained model's performance. In this case, we look for precision and the recall score to validate the model's performance in classifying cats and dogs to assess false positives and true positives to get a realistic understanding of the model's performance. If and when we are satisfied with the results, we can proceed to the next step, or else reiterate the previous steps to get a decent performing model for the pet park image classification service.

- **Model Packaging**: After the trained model has been tested in the previous step, the model can be serialized into a file or containerized (using Docker) to be exported to the production environment.
  Use case implementation:
  The model we trained and tested in the previous steps is serialized to an ONNX file and is ready to be deployed in the production environment.

- **Model registering**: In this step, the model that was serialized or containerized in the previous step is registered and stored in the model registry. A registered model is a logical collection or package of one or more files that assemble, represent, and execute your ML model. For example, multiple files can be registered as one model. For instance, a classification model can be comprised of a vectorizer, model weights, and serialized model files. All these files can be registered as one single model. After registering, the model (all files or a single file) can be downloaded and deployed as needed.
  Use case implementation:
  The serialized model in the previous step is registered on the model registry and is available for quick deployment into the pet park production environment.
  By implementing the preceding steps, we successfully execute the ML pipeline designed for our use case. As a result, we have trained models on the model registry ready to be deployed in the production setup. Next, we will look into the workings of the deployment pipeline.

iii. Deploy

The deploy module enables operationalizing the ML models we developed in the previous module (build). In this module, we test our model performance and behavior in a production or production-like (test) environment to ensure the robustness and scalability of the ML model for production use. Deploy Pipeline has two components – production testing and production release – and the deployment pipeline is enabled by streamlined CI/CD pipelines connecting the development to production environments

- **Application testing**: Before deploying an ML model to production, it is vital to test its robustness and performance via testing. Hence we have the "application testing" phase where we rigorously test all the trained models for robustness and performance in a production-like environment called a test environment. In the application testing phase, we deploy the models in the test environment (pre-production), which replicates the production environment.
  The ML model for testing is deployed as an API or streaming service in the test environment to deployment targets such as Kubernetes clusters, container instances, or scalable virtual machines or edge devices as per the need and use case. After the model is deployed for testing, we perform predictions using test data (which is not used for training the model; test data is sample data from a production environment) for the deployed model, during which model inference in batch or periodically is done to test the model deployed in the test environment for robustness and performance.
  The performance results are automatically or manually reviewed by a quality assurance expert. When the ML model's performance meets the standards, then it is approved to be deployed in the production environment where the model will be used to infer in batches or real time to make business decisions.
  Use case implementation:
  We deploy the model as an API service on an on-premises computer in the pet park, which is set up for testing purposes. This computer is connected to a CCTV camera in the park to fetch real-time inference data to predict cats or dogs in the video frames. The model deployment is enabled by the CI/CD pipeline. In this step, we test the robustness of the model in a production-like environment, that is, whether the model is performing inference consistently, and an accuracy, fairness, and error analysis. At the end of this step, a quality assurance expert certifies the model if it meets the standards.

- **Production release**: Previously tested and approved models are deployed in the production environment for model inference to generate business or operational value. This production release is deployed to the production environment enabled by CI/CD pipelines.
  Use case implementation:
  We deploy a previously tested and approved model (by a quality assurance expert) as an API service on a computer connected to CCTV in the pet park (production setup). This deployed model performs ML inference on the incoming video data from the CCTV camera in the pet park to classify cats or dogs in real time.

iv. Monitor

The monitor module works in sync with the deploy module. Using explainable monitoring, we can monitor, analyze, and govern the deployed ML application (ML model and application). Firstly, we can monitor the performance of the ML model (using pre-defined metrics) and the

deployed application (using telemetry data). Secondly, model performance can be analyzed using a pre-defined explainability framework, and lastly, the ML application can be governed using alerts and actions based on the model's quality assurance and control. This ensures a robust monitoring mechanism for the production system.

- **Monitor**: The monitoring module captures critical information to monitor data integrity, model drift, and application performance. Application performance can be monitored using telemetry data. It depicts the device performance of a production system over a period of time. With telemetry data such as accelerometer, gyroscope, humidity, magnetometer, pressure, and temperature we can keep a check on the production system's performance, health, and longevity.
  Use case implementation:
  In real time, we will monitor three things – data integrity, model drift, and application performance – for the deployed API service on the park's computer. Metrics such as accuracy, F1 score, precision, and recall are tracked to data integrity and model drift. We monitor application performance by tracking the telemetry data of the production system (the on-premises computer in the park) running the deployed ML model to ensure the proper functioning of the production system. Telemetry data is monitored to foresee any anomalies or potential failures and fix them in advance. Telemetry data is logged and can be used to assess production system performance over time to check its health and longevity.

- **Analyze**: It is critical to analyze the model performance of ML models deployed in production systems to ensure optimal performance and governance in correlation to business decisions or impact. We use model explainability techniques to measure the model performance in real time. Using this, we evaluate important aspects such as model fairness, trust, bias, transparency, and error analysis with the intention of improving the model in correlation to business.
  Over time, the statistical properties of the target variable we are trying to predict can change in unforeseen ways. This change is called "model drift," for example, in a case where we have deployed a recommender system model to suggest suitable items for users. User behavior may change due to unforeseeable trends that could not be observed in historical data that was used for training the model. It is essential to consider such unforeseen factors to ensure deployed models provide the best and most relevant business value. When model drift is observed, then any of these actions should be performed:

  – The product owner or the quality assurance expert needs to be alerted.
  – The model needs to be switched or updated.
  – Re-training the pipeline should be triggered to re-train and update the model as per the latest data or needs.

  Use case implementation:
  We monitor the deployed model's performance in the production system (a computer connected to the CCTV in the pet park). We will analyze the accuracy, precision, and recall scores for the model periodically (once a day) to ensure the model's performance does not deteriorate below the threshold. When the model performance deteriorates below the threshold, we initiate system governing mechanisms (for example, a trigger to retrain the model).
- **Govern**: Monitoring and analyzing is done to govern the deployed application to drive optimal performance for the business (or the purpose of the ML system). After monitoring and analyzing the production data, we can generate certain alerts and actions to govern

the system. For example, the product owner or the quality assurance expert gets alerted when model performance deteriorates (for example, low accuracy, high bias, and so on) below a pre-defined threshold. The product owner initiates a trigger to retrain and deploy an alternative model. Lastly, an important aspect of governance is "compliance" with the local and global laws and rules. For compliance, model explainability and transparency are vital. For this, model auditing and reporting are done to provide end-to-end traceability and explainability for production models.

Use case implementation:

We monitor and analyze the deployed model's performance in the production system (a computer connected to the CCTV in the pet park). Based on the analysis of accuracy, precision, and recall scores for the deployed model, periodically (once a day), alerts are generated when the model's performance deteriorates below the pre-defined threshold. The product owner of the park generates actions, and these actions are based on the alerts. For example, an alert is generated notifying the product owner that the production model is 30% biased to detect dogs more than cats. The product owner then triggers the model re-training pipeline to update the model using the latest data to reduce the bias, resulting in a fair and robust model in production. This way, the ML system at the pet park in Barcelona is well-governed to serve the business needs.

IV. Continuous integration, delivery, and deployment in MLOps

Automation is the primary reason for CI/CD in the MLOps workflow. The goal of enabling continuous delivery to the ML service is to maintain data and source code versions of the models, enable triggers to perform necessary jobs in parallel, build artifacts, and release deployments for production. Several cloud vendors are promoting DevOps services to monitor ML services and models in production, as well as orchestrate with other services on the cloud. Using CI and CD, we can enable continual learning, which is critical for a ML system's success. Without continual learning, a ML system is deemed to end up as a failed **Proof of Concept (PoC)**.

Only a model deployed with continual learning capabilities can bring business value.

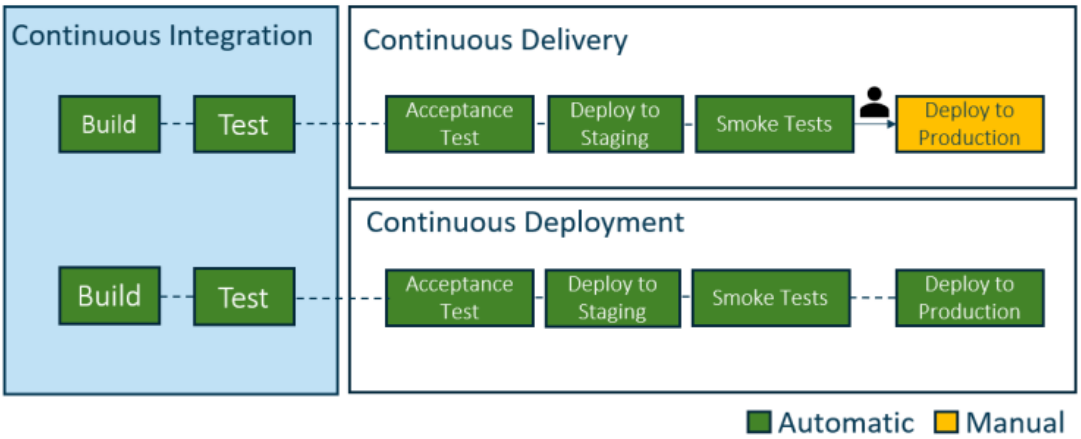


Figure 7: *Continuous integration, delivery, and deployment pipelines.*

i. Continuous integration

CI aims to synchronize the application (ML pipeline and application) with the developer in real time. The developer's changes in commits or merges are validated by creating an application build on the go and by performing automated tests against the build. CI emphasizes automated testing with a focus on checking the application's robustness (if it is not broken or bugged) when new commits are merged to the master or main branch. Whenever a new commit is made to the master branch, a new build is created that is tested for robustness using automated testing. By automating this process, we can avoid delayed delivery of software and other integration challenges that can keep users waiting for days for the release. Automation and testing are at the heart of CI.

ii. Continuous delivery

Continuous delivery extends from CI to ensure that the new changes or releases are deployed and efficiently brought to users; this is facilitated by automating testing and release processes. Automating testing and release processes enable developers and product managers to deploy the changes with one click of a button, enabling seamless control and supervision capabilities at any phase of the process. In the continuous delivery process, quite often, a human agent (from the QA team) is involved in approving a build (pass or fail) before deploying it in production (as shown in Figure 7.1 in a continuous delivery pipeline). In a typical continuous delivery pipeline, a build goes through preliminary acceptance tests before getting deployed on the staging phase where a human agent supervises the performance using smoke tests and other suitable tests. Once the smoke tests have been passed, the human agent passes the build to be deployed in production. Automating the build and release process and having a human agent involved in the process ensures great quality as regards production and we can avoid some pitfalls that may go unnoticed with a fully automated pipeline. Using continuous delivery, a business can have full control over its release process and release a new build in small batches (easy to troubleshoot in the case of blockers or errors) or have a full release within a requisite time frame (daily, weekly, or monthly).

iii. Continuous deployment

CD enables full automation and goes one step further than continuous delivery. All stages of build and release to your production are completely automated without any human intervention, unlike in continuous delivery. In such an automated pipeline, only a failed test can stop a new change from being deployed to production. Continuous deployment takes the pressure off the team to maintain the release pipeline and accelerates deployment straight to the customers enabling continual learning via feedback loops with customers.

With such automation, there is no longer a release day for developers. It takes the pressure off them and they can just focus on building the software without worrying about tests and release management. Developers can build, test, and deploy the software at their convenience and can go live within minutes instead of waiting for release days or for human approval, which can delay the release of software to users by days and sometimes weeks. Continuous deployment ensures full automation to deploy and serve robust and scalable software to users.

V. Overview of ML Deployment Architectures

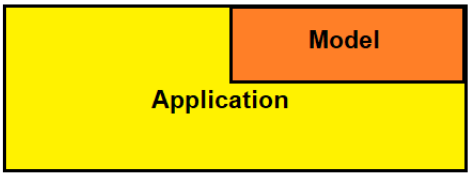
i. Model Embedded in Application

In this scenario, the trained model is embedded in the application as a dependency. For example, we can install the model into the application with a pip installation or the trained model can be pulled into the application at build time from a file storage (i.e. AWS S3).

An example of this instance is if we had a flask application that we used to predict the value of a property. Our Flask application would serve an HTML page which we could use as an interface to collect information about a property a user would like to know an estimated value for. The Flask application would take those details as inputs, forward them to the model to make a prediction then return them to the client.

In the example above, the predictions will be returned to the user's browser, however, we can vary this method to embed the model on a mobile device.

This approach is much simpler than other approaches, but there's a simplicity-flexibility trade-off. For instance, to make a model update the entire application would have to be redeployed (on a mobile device, a new version would need to be released).



Example Embedded Architecture; Image By Author

- **Pre-Trained**: Yes
- **On-the-fly Predictions**: Yes

Figure 8: *Model Embedded in Application.*

ii. Dedicated Model API

In this architecture, the trained machine learning model becomes a dependency of a separate Machine Learning API service. Extending on from the Flask application to predict the value of a property example above, when the form is submitted to the Flask application server, that server makes another call — possibly using REST, gRPC, SOAP, or Messaging (i.e. RabbitMQ) — to a separate microservice that has been dedicated to Machine learning and is exclusively responsible for returning the prediction.

Differing from the embedded model approach, this method compromises simplicity for flexibility. Since we'd have to maintain a separate service there is increased complexity with this architecture, but there is more flexibility since the model deployments are now independent of the main application deployments. Additionally, the model microservice or main server can be scaled separately to deal with higher volumes of traffic or to potentially serve other applications.

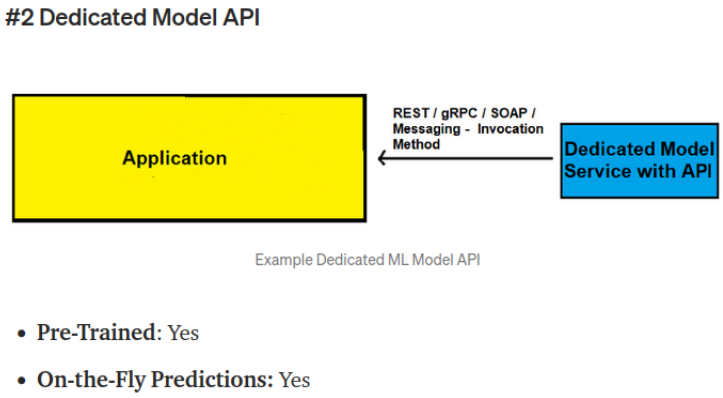


Figure 9: *Dedicated Model API.*

iii. Model Published as Data

In this architecture, our training process publishes a trained model to a streaming platform (i.e. Apache Kafka) which will be consumed at runtime by the application, instead of build time — eligible to subscribe for any model updates.

The recurring theme of simplicity-flexibility trade off occurs here once again. Maintaining the infrastructure required for this archeticutre demands much more engineering sophistication, however ML models can be updated without any applications needing to be redeployed — this is because the model can be ingested at runtime.

To extend on our predicting value of a property example, the application would be able to consume from a dedicated topic from the designated streaming service (i.e. Apache Kafka).

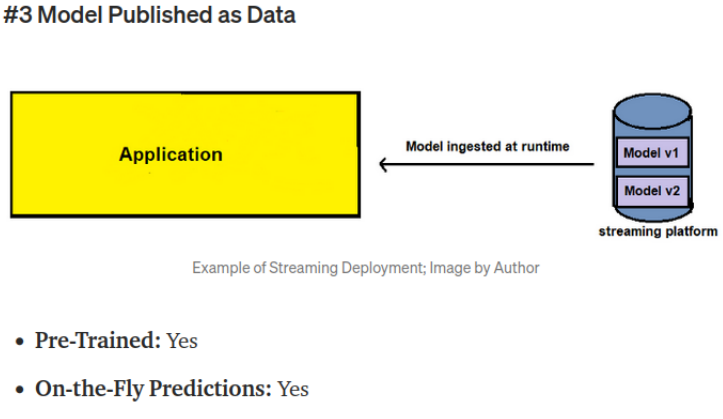


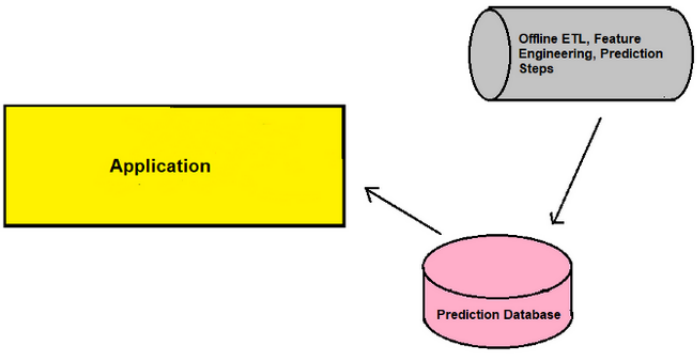
Figure 10: *Model Published as Data.*

iv. Offline Predictions

This approach is the only asynchronous approach we will be exploring. Predictions are triggered and run asynchronisously either by the application or as a scheduled job. The predictions will be collected and stored — this is what the application uses to serve the predictions via a user interface.

Many in industry have moved away from this architecture, but it's much more forgiving in a sense that predictions can be inspected before being returned to a user. Therefore, we reduce the risk of our ML system making errors since predictions are not on the fly.

In regards to the simplicity-flexibility tradeoff, this system compromises simplicity for more flexibility.



Offline Architecture Example; Image by Author

- **Pre-trained:** Yes
- **On-the-fly Predictions:** No

Figure 11: *Offline Predictions.*