**End to End ML Pipeline:**

**What is a Machine Learning Pipeline:**

* A machine learning (ML) pipeline streamlines the steps from data processing to deploying models, making the journey from idea to implementation smoother. Think of it as a well-organized assembly line for ML projects, where each phase has its unique role in transforming data into predictions. Let’s break down the pipeline into three main stages, often referred to as Loops, each focusing on different aspects of ML model development.
* **Inner Loop: Quick Experiments and Model Trials:**
  + The inner loop is all about speed and exploration. Here, you dive into the data, play around with features, and test out various models to see what sticks. It’s a sandbox environment where failing fast is a good thing because it leads you quicker to promising models. This stage is heavy on offline experiments, allowing for rapid iteration without the overhead of deploying models. In this phase, the primary outputs are diverse machine learning models that have been rapidly developed and tested against the dataset. These models are accompanied by initial performance metrics that indicate their effectiveness in handling the given task. The key here is the generation of insights regarding which models, features, and data preprocessing techniques show the most promise for further development.
* **Middle Loop: Fine-Tuning the Contenders:**
  + Once you’ve got promising models from the inner loop, the middle loop is where they get polished. This stage is about refining these models. Tweaking parameters, and validating them more thoroughly. It’s a bit like a boot camp for models, ensuring they’re robust and ready for the real world. The aim here is to bridge the gap between experimental models and those ready for prime time, ensuring only the best candidates make it through. The middle loop produces a smaller set of models that have been optimized for better performance through hyperparameter tuning and more sophisticated validation techniques.
* **Outer Loop: Launching Models into the Real World**
  + The outer loop is where things get real. This is about taking those refined models and deploying them into production environments. It involves integrating the model with existing systems, scaling it up to handle real-world data flows, and setting up monitoring to keep an eye on performance. This stage is where the engineering effort ramps up, focusing on making the model reliable, scalable, and efficient in a live setting. The output of this stage is not just the live model itself but also the infrastructure for continuous monitoring of model performance, data quality, and system health. This ensures the model remains effective over time and adapts to changing conditions or data distributions.
* Diagram of a diagram of a process

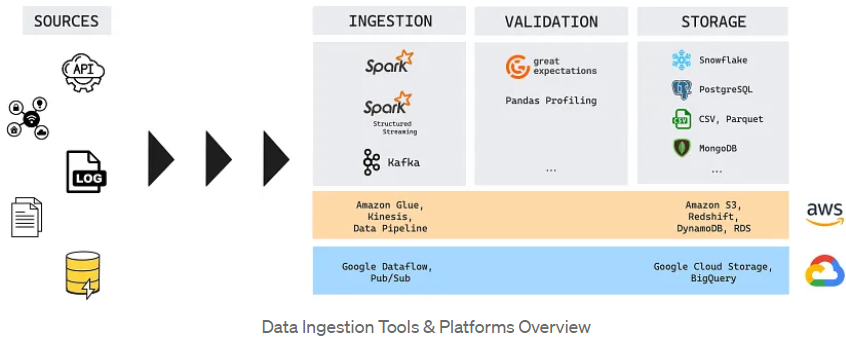
  Description automatically generated
  + Each of the ML Lifecycle inner loops has its own pipeline. We will focus on the outer loop, preferably the Production Machine Learning Pipeline where every step of the pipeline happens automatically and insights from the raw data are reaching the end users predictably

**Production Machine Learning (ML) Pipeline Overview:**

* A production ML pipeline is essentially the backbone of any deployed machine learning system. It’s the automated workflow that ensures your models are not just academic exercises but tools delivering real-world value. This pipeline spans from raw data collection to model deployment and beyond, ensuring that your ML models are continuously improved and aligned with business needs. Here’s a breakdown of what a production-level ML Pipeline typically includes:
  1. **Data Ingestion and Validation:**
     + Starting off, data ingestion and validation are about pulling in data from diverse sources and ensuring it’s clean and structured correctly for ML use. This step is critical because the quality of your input data directly impacts the reliability of your model’s predictions. Think of it as the preprocessing you do to ensure your datasets are free from anomalies or missing values that could skew your model’s learning process.
  2. **Data Preprocessing and Feature Engineering:**
     + In the data preprocessing and feature engineering phase, you’re essentially preparing your dataset for optimal learning. This includes normalizing or standardizing you data, encoding categorical variables, and ingeniously crafting features that could significantly boost your model’s predictive power. IT’s a blend of art and science, where your domain knowledge can lead to the creation of features that make your models understand the nuances of the data better.
  3. **Model Retraining and Versioning**
     + Given the dynamic nature of data, models can’t remain static. Regular retraining with fresh data keeps your models up-to-date and reflective of current trends. Versioning, on the other hand, is like maintaining a detailed logbook of your model iterations, allowing you to track changes and revert to previous versions if a new model underperforms. This ensures your ML system remains adaptable and traceable over time.
  4. **Monitoring and Performance Tracking:**
     + Once live, it’s crucial to keep a vigilant eye on your models with robust monitoring and performance tracking. This isn’t just about watching for model drift or degradation but also ensuring the operational health of your ML system. Tools and frameworks that offer real-time monitoring of model performance metrics and system logs are indispensable here. They help you maintain the integrity and performance of your ML solutions in the face of evolving data distributions.
* **A diagram of a production machine learning

  Description automatically generated**

**Data Ingestion and Validation:**

* When setting up machine learning workflows, the initial steps of data ingestion and validation are essential. These processes ensure that the data, whether coming from **IoT sensors, logs, APIs, or databases**, is ready for the heavy lifting that comes later in feature processing and model training.
* **Gathering Data from Varied Sources:**
  + The variety of data sources presents both an opportunity and a challenge. Tools provided by cloud services, such as AWS kinesis for streaming data and Google Dataflow for batch and stream processing, simplify the task of pulling in data at scale. Similary, open-source solutions like Apache Spark for batch processing and Kafka for handling real-time data streams offer flexibility and scalability, making it easier to kickstart the data pipeline.
* **The Importance of Data Quality:**
  + Before moving on to the more glamorous task of model training, it’s necessary to ensure that the data is in good shape. This means checking for NULL or missing values, ensuring consistency across data types and formats, validating that the data falls within acceptable ranges, and confirming that there are no duplicate records. Validating data upfront saves time and headaches later by avoiding “garbage-in-garbage-out” scenarios in model training. For this, Great Expectations and Deequ offer automated ways to set and check data quality criteria, making these checks more efficient. For a deeper dive into data quality, Python libraries like “Pandas Profiling” or others provide detailed reports that highlight potential issues directly from your Jupyter notebook.
* **Where to Store Your Data:**
  + Once data is ingested and validated, choosing the right storage is crucial for further processing. Cloud storage options like **AWS S# or Google Cloud Storage** are versatile choices for keeping files in formats like **Parquet, JSON, or CSV**. For structured data, cloud data warehouses such as **Snowflake, AWS Redshift, or Google BigQuery** offer powerful and scalable querying capabilities. Meanwhile, NoSQL options like **MongoDB** or **AWS DynamoDB** are great for handling unstructured or semi-structured data, offering flexibility and scalability.
* **Keeping Your Data Pipeline Smooth:**
  + Automation is key in managing data pipelines effectively. It minimizes manual effort and helps maintain accuracy and efficiency. Implementing data versioning with tools like **DVC** can also greatly enhance the manageability of your data, ensuring that you can track changes and maintain consistency across your ML projects.
* To summarize, setting up a reliable data ingestion and validation pipeline might not be the most glamorous part of machine learning, but it’s foundational. By selecting the right tools for the job and following best practices, you can ensure that your data is in the best shape by the time it’s ready for model training, paving the way for more successful and stress-free model development.
* ****

**Data Preprocessing and Feature Engineering:**

* Data Preprocessing and Feature Engineering comprise the steps that transition data from your raw or intermediary (validated) data sources into features, ready for model training and inference. These transformations are important to be performed at this stage in the ML pipeline, ideally on distributed infrastructure in order to make sure the model training phase has the input data readily available and in a proper format that would necessitate minimal manipulations during the training pipeline.
* **Data Preprocessing:**
  + The goal of data preprocessing is to clean and prepare data for analysis and modeling. This involves:
    - Normalizing or standardizing numerical values for consistency.
    - Encoding categorical variables to make them interpretable by ML algorithms.
    - Filtering out outliers and correcting errors to enhance data quality.
    - Converting data types to meet algorithm requirements.
* **Featurization:**
  + Feature engineering uses domain knowledge to extract and create meaningful features from raw data, improving model performance. This process can include:
    - Dimensionality reduction (e.g., Principal Component Analysis) to reduce data complexity.
    - Creating new features by manipulating existing ones to uncover relevant information.
    - Applying techniques for time-series data, like aggregations and window functions, to capture temporal dynamics.
* **Data Engineering Infrastructure:**
  + - This stage often utilizes infrastructure similar to that used for data ingestion, involving data processing pipelines and storage. “Feature stores” are commonly integrated into this infrastructure, offering a centralized place to manage features for both batch processing and real-tome applications. These stores vary in capability; some manage features metadata and storage, while others, like Qwak’s Feature Store, also handle data ingestion and transformations.
    - For example, connecting to a Kafka source to read sensor data involves steps that can be simplified using a feature store. Instead of manually managing connections and queries, a feature store can streamline data ingestion form various sources, enhancing the efficiency of creating features for models like Anomaly Detection.
    - It’s increasingly common to see data ingestion, processing, and feature engineering converge within a feature store’s ecosystem. This integration simplifies the pipeline, allowing for a more streamlined flow from raw data to prepared features, ready for machine learning applications.
    - Feature stores equipped with built-in capabilities for data ingestion and processing significantly reduce the complexity associated with connecting to and querying data sources. Consider the process of accessing data from AWS Redshift. Traditionally, this would involve multiple steps: authenticating via AWS’s boto3 Python client, retrieving database credentials, and then executing queries using libraries like psycopg2. In contrast, a feature store with Redshift integration simplifies this to merely providing an IAM Role ARN for authentication along with the SQL query or table name. This level of abstraction is especially beneficial when dealing with diverse data sources, enabling seamless data joining and aggregation.
    - Expanding on data ingestion, utilizing a feature store directly connected to raw data sources streamlines the entire process. Instead of the traditional multi-stage approach-where data moves from raw input to validated intermediary storage and finally to processing and featurization-the feature store handles both ingestion and processing efficiently in a single step.
  + **Below is Qwak’s Feature Store:**
  + from qwak.feature\_store.data\_sources.streaming.kafka import KafkaSource  
    from qwak.feature\_store.data\_sources.streaming.kafka.deserialization import GenericDeserializer, MessageFormat   
      
    # Define a deserializer for sensor data   
    # Assuming sensor data is formatted in JSON and adheres to a specific schema   
    deserializer = GenericDeserializer(  
     message\_format=MessageFormat.JSON,  
     schema=""   
    )   
      
    # Configure the Kafka source for sensor data ingestion   
    # The source is tailored to subscribe to a topic dedicated to sensor data   
    sensor\_kafka\_source = KafkaSource(  
     name="sensor\_data\_source",  
     description="Sensor Data Source for Real-Time ML Pipeline",  
     bootstrap\_servers="broker.sensor.network.com:9094, broker2.sensor.network.com:9094",  
     subscribe="sensor\_data\_topic",  
     deserialization=deserializer   
    )   
      
    print(sensor\_kafka\_source.get\_sample(10))
    - The code snippet above demonstrates how to set up the Feature Store to connect to a Kafka source and retrieve sensor data. This data is intended for further transformation into features suitable for an Anomaly Detection model
  + **Handling Different Data:**
    - Data originates from diverse sources and can be categorized based on its structure; it can either be **structured**, deriving from sources with a fixed schema such as relational databases, or **unstructured**, coming form NoSQL databases like MongoDB which are document-based, or even from raw text that requires more complex manipulations like embedding. Specifically, for embedding raw text into usable formats for machine learning, “Vector Databases” provide efficient solutions for storing and retrieving these embedded vectors during training and inference phases.
    - From the perspective of data flow or availability, sources can be categorized as **static** or **streaming**. Static data sources, which do not change once stored, are generally processed in batches through scheduled jobs. On the other hand, streaming sources, also referred to as dynamic data, are continuously generated by activities such as sensors, logs, or user interactions, and require processing in real-time or near-real-time. Tools designed for managing streaming data include Apache Flink or Spark Streaming, among others, to facilitate timely data processing.
    - Major cloud providers offer a range of services that cater to both structured and unstructured data, as well as static and streaming data sources. These services, primarily focused on data pipelines, facilitate the movement and transformation of data into feature storage but are usually distinct from the feature store itself. Notable examples of theses services include Databricks, AWS Glue/EMR, which are powered by Spark, and Google Cloud’s Dataflow. These tools are integral to the data engineering process, ensuring data is in the right format and place for subsequent machine learning tasks, although they operate separately from the feature store, which centralizes the management of features for machine learning models.
  + **Production Considerations:**
    - **Versioning:** Implementing tools like DVC for versioning data, preprocessing pipelines, and features is crucial for reproducibility and rollback capabilities. DVC works by creating lightweight metadata files that reference the actual data files stored separately. It applies the principles of version control (similar to Git) to data and machine learning models.
    - **Automation:** Automating the processing of new data through scheduled jobs or triggers based on specific conditions ensures timely updates and model relevance.
    - **Consistency:** Maintaining consistency between training and serving data prevents training-serving skew, with Feature Stores playing a key role in this aspect.
    - **Monitoring:** Continuously monitoring for shifts in data distribution or quality (data drift) is essential, as it may indicate the need for adjustments in the preprocessing and feature engineering steps.
    - In environments where ML models are deployed for real-time predictions, the capacity to store and retrieve features with minimal latency is indispensable. This requirement emphasizes the importance of a well-architected Feature Store that supports the dynamic requirements of real-time ML applications.
  + To continue the example outlined above, we’ll use the ingested data from the sensor\_data\_source which is transformed into features for our Anomaly Detection model and stored them in the Qwak Offline and Online Feature Stores, to be sued for training and inference:
  + from qwak.feature\_store.feature\_sets import streaming   
    from qwak.feature\_store.feature\_sets.transformations import SparkSqlTransformation, Schema, Column, Type   
    from qwak.feature\_store.feature\_sets.transformations.functions import qwak\_pandas\_udf   
    import pandas as pd   
      
    # Transformation: Calculate the moving average   
    @qwak\_pandas\_udf(output\_schema=Schema([Column(type=Type.double)]))   
    def moving\_average(column\_a: pd.Series, window\_size: int = 3) -> pd.Series:  
     return column\_a.rolling(window=window\_size).mean()   
      
    # Transformation: Calculate the difference from the moving average   
    @qwak\_pandas\_udf(output\_schema=Schema([Column(type=Type.double)]))   
    def diff\_from\_moving\_avg(column\_a: pd.Series, column\_b: pd.Series) -> pd.Series:  
     return column\_a - column\_b   
      
      
    @streaming.feature\_set(  
     name='sensor-data-fs',  
     key='sensor\_id',  
     data\_source=['sensor\_data\_source'],  
     timestamp\_column\_name='timestamp'   
    )   
      
    def transform():  
     return SparkSqlTransformation(  
     sql="""SELECT timestamp,  
     sensor\_id,   
     value AS raw\_value,   
     MOVING\_AVERAGE(value, 3) OVER (PARTITION BY sensor\_id ORDER BY timestamp) AS moving\_avg,   
     DIFF\_FROM\_MOVING\_AVG(raw\_value, moving\_avg) AS diff\_moving\_avg   
     FROM sensor\_data\_source""",   
     functions=[moving\_average, diff\_from\_moving\_avg]   
    )
  + Together, these transformations prepare the streaming sensor data for anomaly detection by creating features that reflect both the normal trends and the deviations that could signify potential issues. The moving average provides a baseline of expected behavior, while the difference from the moving average pinpoints where the data diverges from this expectation, which is essential for identifying anomalies.
  + In the next section, we’ll use the Offline Features from this features set to train and predict sensor anomalies with our Anomaly Detection ML model

**Model Retraining and Versioning**

* The transition from developing and training machine learning models in a research or development setting, often done in notebooks, to deploying these models in a production environment, requires a fundamental shift in approach. This shift is not just in terms of the tools and technologies used but also in the mindset and processes involved. In this section, we delve into the intricacies of automating Continuous Integration (CI) pipelines for model retraining in production, underscoring the differences and requirements that distinguish production-grad model retraining from its development counterpart.
* **Why Retrain?**
  + Before diving into the “how,” it’s worth revisiting the “why.” Models get stale like last week’s bread. Here’s why:
  + **Data Drift:** Imagine if your model was trained on summer clothing preferences, and now it’s winter. People’s choices have shifted, and your model needs to catch up.
  + **Concept Drift:** Sometimes, it’s not just about preferences changing with the season but about the underlying dynamics shifting. Maybe a new fabric becomes all the rage, altering purchase behaviors.
  + **Operational Changes:** The business decides to pivot, targeting a new demographic or introducing a new product line. Your model needs to align with these new objectives.
  + **Feedback Loop Integration:** Every interaction with a user is a learning opportunity. Integrating this feedback keeps the model in tune with user needs and behaviors.
* **A diagram of a window

  Description automatically generated**
* **Automating the CI Pipeline for Model Retraining**
* The automation of a model retraining pipeline in a production setting involves several stages:
  + **Provisioning Training Infrastructure:** Depending on the dataset and the complexity of the model, this could range from a single instance to distributed training across multiple nodes to handle large datasets efficiently
  + **Fetching the Training Docker Image or Scripts:** This step ensures that the training environment is consistent, reproducible, and can be scaled across different environments.
  + **Building the Environment:** This includes initializing the model, fetching the dataset, and running the fitting function. It’s a step where the actual training process begins, leveraging the infrastructure provisioned earlier
  + **Evaluating Model Performance:** Beyond assessing predicting quality, it’s crucial to evaluate the model’s performance in terms of inference timeliness, especially for applications requiring real-time predictions.
  + **Publishing Metrics, Parameters, Artifacts:** This final step involves logging the training outcomes to a Model Registry, facilitating version control, performance tracking, and compliance with audit requirements.
* **A diagram of a training pipeline

  Description automatically generated**
* To continue our Anomaly Detection model example, we’ll fetch the last day’s sensor data and train the model.
* from qwak.model.base import QwakModel   
  from qwak.feature\_store.offline import OfflineClientV2   
  from qwak.feature\_store.offline.feature\_set\_features import FeatureSetFeatures   
  from sklearn.ensemble import IsolationForest   
  from datetime import datetime, timedelta   
  import pandas as pd   
  import qwak   
    
    
  class SensorAnomalyDetectionModel(QwakModel):   
    
   def \_\_init\_\_(self):   
    
   # Initialize the Isolation Forest model   
   self.model = IsolationForest(n\_estimators=100, contamination='auto')  
    
  # Called in the Model Build CI   
  def build(self):   
    
   # Specify the features to fetch from the Feature Store   
   selected\_features = FeatureSetFeatures(   
   feature\_set\_name='sensor-data-fs',   
   feature\_names=['moving\_avg', 'diff\_moving\_avg']   
   )  
    
   # Initialize the Offline Store client   
   offline\_client = OfflineClientV2()   
    
   # Fetch data from the offline feature store since yesterday, considering daily retraining   
   training\_data = offline\_client.get\_feature\_range\_values(   
   features=selected\_features,   
   start\_date=datetime.now() - timedelta(days=1),   
   end\_date=datetime.now()   
   )   
    
   X = training\_data[['moving\_avg', 'diff\_moving\_avg']] # Features for anomaly detection   
    
   # Train the model on the entire dataset   
   self.model.fit(X)   
    
   # This is more of a placeholder as Isolation Forest's main parameter is 'n\_estimators'   
   qwak.log\_param("n\_estimators", self.model.get\_params()['n\_estimators'])
* In the Qwak model’s build process, the build method is called within the CI pipeline to train the model. After training, the model is saved (or “pickled”) into an artifact. This artifact is then stored inside a container image, ready for deployment. To get the features needed for training, we use the OfflineClientV2 to connect to the Feature Store and fetch the features we talked about earlier.
* We’re still working on the QwakModel. Next up, we’ll add the parts that handle starting the model at the deployment and inference logic, which we’ll cover in the section on Model Deployment and Serving.
* For keeping the model up-to-date by retraining and rebuilding it with new data, you can automate the process. You might use Qwak Automation for this, or other tools familiar to many in the ML community, like GitHub Actions, Jenkins, or Apache Airflow
* **Retraining Strategies**
* Retraining strategies in production environments can be broadly categorized into:
  + **Scheduled Retraining:** Regularly retraining the model at fixed intervals regardless of whether drift is detected. This approach is straightforward but may not be the most efficient.
  + **Performance-Based Retraining**: Triggering retraining sessions based on the model’s performance metrics falling below certain thresholds. This strategy is more dynamic and can be more efficient in utilizing resources.
  + **Continuous Learning**: An advanced strategy where the model is constantly updated with new data, ideally in real-time or near-real-time. This approach requires a sophisticated infrastructure and process to ensure model stability and performance.
* **A screenshot of a computer

  Description automatically generated**

**Model Deployment and Serving:**

* In a production Machine Learning pipeline, the journey from a trained model to one that actively contributes to decision-making involves two stages: deployment and serving. Deployment is the process of integrating the trained model into a production environment, making it ready to handle real data and start delivering insights. Serving, meanwhile, refers to the model’s operational phase, where it processes incoming data and provides predictions or analyses in real-time, batch, or streamlining modes.
* **Model Deployment Strategies:**
* Understanding the deployment landscape is like selecting the right vehicle for your journey, with each mode-batch, real-time, or asynchronous (streaming)-having its domain where it excels.
  + **Batch processing:** Is the best visualized as the long-haul truck of model deployments, prioritizing capacity and efficiency over speed. It’s the strategy of choice when dealing with large data volumes that do not require immediate processing. Technically, batch processing refers to running your machine learning model on a complete set of data at once, at scheduled intervals. This method doesn’t require the model to be loaded and ready to predict at all times. Instead, data is collected over a period (e.g., a day, week, or month), and then the model processes this data in one large “batch”. The model’s predictions, once generated, are often stored for later use or further analysis. Batch processing is commonly used for applications where real-time decision-making is not critical, such as generating nightly reports, performing daily analytics, or updating recommendation systems based on user activity collected throughout the day.
    - **Deployment:** The model is typically integrated into a scheduled workflow, such as a daily or weekly job, where it waits inactive until triggered. The infrastructure for batch models doesn’t need to be constantly running, which can be cost-effective, especially in cloud environments where resources can be scaled up or down as needed.
    - **When to Choose Batch:** Opt for batch processing when efficiency and volume take precedence over speed, such as for analytics or operations that can be queued for off-peak processing.
    - **Challenges:** Managing job scheduling and data throughput efficiently is key.
  + **Real-time predictions:** are the speedy delivery bikes, designed for agility and immediate delivery. In real-time processing, the model is deployed in a way that it can receive and predict on data instantly, often within milliseconds or seconds. This deployment requires the model to be hosted in an environment that keeps it loaded and ready to make predictions as soon as new data arrives. Real-time processing is essential for applications where immediate action based on the model’s predictions is necessary, such as fraud detection in financial transactions, real-time personalization on web platforms, or immediate content recommendation.
    - Another aspect of real-time model deployment is that it functions through an endpoint, much like a web service in software applications. This endpoint is the interface where data is sent for prediction and from where the model’s output is received. Managing this endpoint requires a Continuous Delivery (CD) pipeline tailored to ML deployments. This pipeline automates the process of updating the model, testing it with new data, and ensuring it’s performing optimally before and after deployment to the production environment.
    - **A/B Testing:** Before deploying a new model to production, it’s common practice to run A/B tests. This means comparing the new model’s performance against the current one on a segment of live traffic to ensure it actually improves upon the existing setup.
    - **Automated Rollback:** If the newly deployed model underperforms or causes issues, an automated rollback mechanism can revert to the previous model version. This ensures stability and maintains performance standards, minimizing the impact of any deployment-related problems.
    - **When to Choose Real-Time:** This is the go-to strategy when your application demands instant responses and operational efficiency relies on the ability to process data on-the-fly.
    - **Challenges:** Requires meticulous testing and scalability solutions to handle live traffic without latency issues.
  + **Asynchronous (Streaming):** deployments offer a balanced route, like a city bus, handling continuous data flow with relatively quick but not instant processing. It’s well suited for scenarios where data arrives steadily, and there’s a need for somewhat timely processing without the strict immediacy of real-time systems. It’s a method where the model processes data as it arrives, but not necessarily instantly. Data flows into the system continuously, and the model processes it in small, manageable batches. This setup allows for near-real-time predictions with a slight delay, accommodating the time it takes to collect a small batch of data. Streaming is particularly useful for applications that need to process data in real-time but can tolerate a short delay (e.g., processing live social media feeds, monitoring IoT device streams, or real-time analytics where a lag of a few seconds to a minute is acceptable).
    - **When to Choose Streaming:** Streaming is your pick when dealing with continuous data flows that require faster processing than batch can offer but do not necessitate the immediacy of real-time predictions.
    - **Challenges:** Balancing latency and throughput, ensuring the system can keep pace with incoming data flows by keeping an eye on the producer/consumer lag.
* A diagram of a process

  Description automatically generated
  + In the code snippet provided, we show how the inference part of our sensor Anomaly Detection model works. This is part of the QwakModel class we talked about before and is included in the model’s build process. Here’s how it functions: when the model gets a sensor ID from an HTTP request, it automatically looks up the corresponding features using that ID. Then, it performs inferences with those features and sends back an “anomaly\_score.” This process enables the model to assess new sensor data in real-time and determine if there’s an anomaly
* class SensorAnomalyDetectionModel(QwakModel):   
  # Called in the Model Build CI   
  def build(self):  
   pass # Defined above   
    
  # Called when the model is deployed for inference   
  def initialize\_model(self):   
  pass   
    
  # Model inputs and outputs schema containing ID from the request and features from the Feature Store   
  def schema(self) -> ModelSchema:  
   return ModelSchema(  
   inputs=[  
   RequestInput(name='sensor\_id'),  
   FeatureStoreInput(name='sensor\_data\_fs.moving\_avg'),  
   FeatureStoreInput(name='sensor\_data\_fs.diff\_moving\_avg')  
  ],  
  outputs=[InferenceOutput(name="anomaly\_score", type=float)] )   
    
  # Inference logic, feature extraction enabled means it automatically extracts features for given sensor ID from the Online Store   
  @qwak.api(feature\_extraction=True)   
  def predict(self, request\_df: pd.DataFrame, extracted\_df: pd.DataFrame) -> pd.DataFrame:  
   # Prepare features for prediction  
   X = extracted\_df[['moving\_avg', 'diff\_moving\_avg']]   
    
   # Perform anomaly detection   
   prediction = self.model.predict(X)  
    
   # Convert prediction to DataFrame and  
   return (pd.DataFrame(prediction, columns=["anomaly\_score"]))
  + This model can be deployed on Qwak as either a **real-time, batch, or streaming model**. For ease of explanation, let’s say we deploy it as a real-time REST endpoint using the Qwak CLI
* qwak models deploy realtime - model-id sensor\_anomaly\_detection - instance 'medium' - variation-name default
  + Once deployed, you can test the model using a REST client like curl. Here’s how you might send a request to the model’s endpoint, using a sensor ID in the request body:
* curl - location - request POST 'https://models.acme\_sensors.qwak.ai/v1/sensor\_anomaly\_detection/predict' \   
  - header 'Content-Type: application/json' \   
  - header 'Authorization: Bearer $QWAK\_TOKEN' \   
  - data '{"columns":["sensor\_id"],"index":[0],"data":[["a8f5f167f44f4964e6c998dee827110c"]]}'
  + The response from the model would look something like this:
* {   
  "columns": ["anomaly\_score"],  
  "index": [0],  
  "data": [[0.85]]  
  }
  + This response indicates that there’s a high likelihood of an anomaly associated with the provided sensor ID, suggesting the need for further investigation or action.

**Model Monitoring and Performance Tracking:**

Model monitoring and performance tracking are important for maintaining the effectiveness of machine learning models over time. Having detailed inference data, which includes both the inputs to the model and its outputs, allows for ongoing monitoring and comparisons with initial training data.

* **Data Monitoring:**
  + Data monitoring focuses on observing changes in the distribution of data and its quality. Shifts in data distribution indicate that the real-world data the model encounters may have changed since its training, which can affect performance. Similarly, any decrease in data quality, such as missing values or incorrect inputs, can impact the model’s output. Regular monitoring helps ensure the data remains suitable for the model’s needs.
  + Data monitoring should begin as soon as data collection starts and continue throughout the data preparation phase. It’s essential during the initial data gathering, as well as ongoing as new data is collected or generated over time. This ensures the data fed into the model during training and inference is of high quality and representative. Data monitoring involves.
    - Tracking changes in data distributions, such as shifts in mean or variance.
    - Identifying missing values, outliers, or increases in data errors and noise.
    - Monitoring for new, unseen categories in categorical data.
  + A screenshot of a computer

    Description automatically generatedThe output of a data monitoring system is an alert for data scientists to investigate the data anomalies. It may trigger data preprocessing adjustments or reevaluation of feature engineering.
* **Model Monitoring:**
  + Model monitoring involves tracking the model’s performance to detect any degradation. Over time, a model may not perform as well due to changes in underlying data patterns. Training-serving skew, where the production data diverges from the training data, is another factor that requires attention. Routine performance checks and model updates are necessary to keep the model aligned with current data trends.
  + Model monitoring primarily occurs post-model deployment into a production environment. However, setting up the groundwork for effective model monitoring happens earlier during the model development, training, and validation. By establishing baselines and expectations during these stages, you create reference points for comparison once the model is live. Model monitoring involves:
    - Tracking model performance metrics like accuracy, precision, recall, or F1 score over time.
    - Detecting drifts in model predictions due to changes in underlying data patterns.
    - Identifying training-serving skew.
  + A screenshot of a computer

    Description automatically generatedThe output of a model monitoring job is an alert (trigger) that may lead to model retraining with updated data, tweaking model hyperparameters, or developing new features that better capture current user behavior and preferences
* **Serving Monitoring:**
  + Serving monitoring is about overseeing the model’s deployment infrastructure. This includes checking the latency, throughput, and error rates of model serving endpoints to confirm that they meet operational standards. Monitoring the serving layer helps identify any deployment issues early, ensuring the model remains available and performs well under different loads.
  + Serving monitoring is specific to the model deployment and serving phase. It starts as soon as the model is deployed for inference in the production environment and continues throughout the model’s lifecycle. Monitoring the serving infrastructure ensures that the model is accessible and performs predictively under operational loads.

A screenshot of a graph

Description automatically generated**Summarizing ML Pipelines:**

* In this guide, we walked through building an end-to-end machine learning (ML) pipeline, focusing on transforming raw data into actionable insights through deployed ML models. Here’s quick overview of what we covered:
* **Data Ingestion and Validation:** Ensuring the data is clean and correctly formatted for ML use.
* **Data Preprocessing and Feature Engineering:** Transforming data into a format that ML models can easily use, including creating new features that improve model performance.
* **Model Retraining and Building:** Keeping models updated with new data to maintain their accuracy and relevance.
* **Model Deployment and Serving:** Making models available in production environments to start providing real-world value, with different strategies like real-time, batch, and streaming.
* **Model Monitoring and Performance Tracking:** Keeping an eye on deployed models to ensure that they perform well and remain reliable over time.
* We also touched on the importance of automating these processes for efficiency and scalability. While we used tools like Qwak for examples, the concepts apply broadly across different platforms and technologies. This foundation allows ML engineers and data scientists to effectively deploy and manage ML models, ensuring they deliver continuous value.

Yes, developing a Retrieval-Augmented Generation (RAG) based chatbot that references company documents using a language model like LLaMA2 would benefit greatly from having an ML pipeline, especially if you plan to introduce new batches of documents regularly.

Here are some key reasons why an ML pipeline would be beneficial in this scenario:

1. **Data ingestion and preprocessing**: An ML pipeline can automate the process of ingesting new batches of documents, cleaning and preprocessing the text data, and preparing it for indexing and retrieval.
2. **Retrieval index updates**: As new documents are added, the retrieval index (e.g., vector database or document store) used by the RAG model needs to be updated. An ML pipeline can handle this process efficiently, ensuring that the retrieval component always has access to the latest document corpus.
3. **Model retraining and evaluation**: Introducing new documents may require retraining or fine-tuning the language model component of the RAG system. An ML pipeline can orchestrate the training process, manage experiment tracking, and evaluate the performance of the updated model using appropriate metrics.
4. **Versioning and reproducibility**: By capturing and versioning all the components involved (data, code, models, configurations), an ML pipeline ensures that the RAG system's development is reproducible and traceable, which is essential for collaboration and auditing purposes.
5. **Deployment and monitoring**: Once a new version of the RAG model is trained and evaluated, the ML pipeline can handle the deployment process, including model serving, load testing, and monitoring the system's performance in production.
6. **Automated updates**: With an ML pipeline in place, you can schedule periodic updates to the RAG system, automatically triggering the ingestion of new document batches, model retraining, and deployment of the updated system.
7. **Scalability**: As the document corpus and user traffic grow, an ML pipeline can help scale the RAG system by distributing the workload across multiple compute resources, enabling parallel processing of data and model training.

By introducing an ML pipeline, you can ensure that your RAG-based chatbot remains up-to-date and performs optimally as new information becomes available, while also maintaining a structured and reproducible development process.