Solution Architecture Overview

This architecture diagram represents a machine learning pipeline running in a **local environment**, from data ingestion to model deployment. The workflow is structured into three main stages: **Build**, **Train**, and **Deploy**, which handle the processes of data ingestion, model training, and real-time deployment of predictions, respectively.

1. Data Ingestion (Beginning of Workflow)

1. **Data Source (Step 1)**:
    - The data for training and inference is stored locally in formats like **CSV** or **Excel** files, which act as the raw data input.
    - This data can be sourced from an **On-Premise Environment** connected via **Direct Connect**.
2. **GroundTruth (Step 2)**:
    - **GroundTruth** is used for data labelling. This step ensures that the raw data is prepared with the necessary labels, allowing it to be used effectively in supervised machine learning.
    - Once labelled, the data is stored in a local storage bucket for further processing.
3. **Data Ingestion API (Step 3)**:
    - A **FastAPI** instance running locally acts as the **Data Ingestion API**, which allows the labelled data to be ingested into the system.
    - This API, hosted on **localhost**, facilitates the secure transfer of data to the storage location for training.
4. **On-Premise Models (Step 5)**:
    - The system may incorporate pre-trained models stored in the **On-Premise Environment**, which are transferred to the local storage when needed.

2. Build Stage (Source Code and CI/CD Pipeline)

The **Build** stage automates the source code management and model preparation pipeline.

1. **Jupyter Notebook (Step 10)**:
    - **Jupyter Notebook** serves as the primary environment for data exploration, feature engineering, and initial model experimentation.
    - Data scientists use it to interact with source data and prototype machine learning models before moving to production.
2. **Code Repository and Pipeline (Steps 11, 12, 13)**:
    - **CodeCommit**: The source code for the model, data processing, and pipeline configuration is stored here, providing version control and collaboration capabilities.
    - **CodePipeline**: Automates the integration and deployment processes, orchestrating the code through various stages.
    - **CodeBuild**: Compiles the code, performs tests, and prepares the environment for the next stages of training and deployment.

Note: This Build Flow is shown in the diagram as **dashed purple lines**.

3. Train Stage (Model Training and Processing)

The **Train** stage involves setting up, training, and testing the machine learning model using the ingested data.

1. **Local Environment (Step 4)**:
   - The **Local Environment** container represents the computing environment where model training and testing occur, using resources on the local machine.
   - The ingested data is processed here for training, and any pre-trained models from the On-Premise Environment may also be integrated.
2. **Model Training Trigger API (Step 6)**:
   - A **FastAPI** endpoint (localhost) serves as a trigger for initiating the model training process.
   - This API enables automated or manual control over the training process, allowing flexibility in model iteration.
3. **Model Training Process (Steps 7 and 8)**:
   - The model is trained using the processed data and tested to validate its performance.
   - Model artefacts, which include trained weights and metadata, are stored locally in the **Model Artifacts** bucket, making them accessible for deployment.

Note: The Training Flow is represented in the diagram by **solid yellow lines**.

4. Deploy Stage (Model Deployment and Real-Time Inference)

The **Deploy** stage includes deploying the trained model, monitoring its performance, and making real-time predictions accessible to the end-user.

1. **Deploy Model and Model Monitor (Step 14)**:
   - The trained model is deployed into a local deployment environment, where it is exposed for real-time predictions.
   - **Model Monitor** tracks the performance of the deployed model, alerting to any data drift or anomalies over time.
2. **Real-Time Prediction API (Step 9)**:
   - A **FastAPI** instance (running on localhost) is set up as the **Real-Time Prediction API**. This endpoint serves incoming prediction requests in real time.
   - The API directly handles user requests through the **API Gateway** and forwards them to the deployed model for predictions.
3. **Streaming API and Real-Time Output Storage**:
   - The **Streaming API** handles continuous data input, allowing for real-time inference without delays.
   - **Real-Time Output Storage** stores the prediction results, enabling logging and analysis of predictions over time.

Note: The Interface Flow in this stage is shown in **dotted green lines**.

4. **API Gateway and User Interface (Step 14)**:
   - **API Gateway** provides a standardised interface for external applications to interact with the Real-Time Prediction API.
   - The **User Interface** component represents the frontend or application where end-users access the predictions, completing the workflow from data ingestion to model prediction.