DS - AI ML Feature Storage

I want to build project feature storage for AI ML model:
- System can handle both batch process (local files) and streamming process (for example stream data, please build function to get mock events from local file, then push event to DB, then use debezium to read CDC, get streamming event)
- datalake build all local hosted (MinIO) and Google cloud datalake, build both config then can select one of them
- dataware house i want to use bigquery
- tech stack: Pyspark, Flink, Kafka, Airflow, Feast, Docker, Kubernetes (GKE), CICD, DBT, Greate expectations,

Firstly, Describe project system and give me a folder structure for best practice

> **Shaping the architecture**
>
> I'm defining the AI ML model storage system, focusing on batch and streaming processes. The architecture includes mock event handling, CDC with Debezium, and integrating MinIO, Google Cloud, and BigQuery. The tech stack is set.

Below is a high-level overview of the system architecture, followed by a recommended folder structure adhering to best practices. The focus is on enabling both batch processing (local file ingestion) and streaming (via Debezium + Kafka), with an option to store data either locally (MinIO) or on Google Cloud (GCS), then ultimately landing data in a BigQuery warehouse, with transformations and validations managed by tools like dbt and Great Expectations.

---

1. System Architecture Overview

1. **Data Ingestion (Batch)**
   - **Local Files → Pyspark**: You periodically load large files (CSV, Parquet, etc.) from local storage.
   - **Data Lake (MinIO or GCS)**: After Spark processing or minimal transformations, you land your data in either local MinIO (on-prem S3-compatible) or Google Cloud

Storage.

- **Orchestration**: Use Airflow to schedule and manage these batch jobs.

2. **Data Ingestion (Streaming)**
   - **Mock Events → Database (e.g., PostgreSQL or MySQL)**: You simulate real-time event data by inserting them into a transactional DB.
   - **Change Data Capture (Debezium)**: Monitors the DB for changes (inserts/updates/deletes) and publishes change events to Kafka topics.
   - **Kafka + Flink (or Spark Streaming)**: Consumes from the Kafka topics, performs streaming transformations/aggregations, and writes the results to the data lake (MinIO/GCS) or directly to the feature store if needed in near real time.
   - **Orchestration**: Airflow (or another scheduler) can trigger these streaming jobs or manage them via longer-running processes.
3. **Data Lake**
   - You maintain configurations for both local S3-compatible (MinIO) and Google Cloud (GCS) so the system can seamlessly switch between local and cloud-based storage depending on the deployment scenario.
4. **Data Warehouse (BigQuery)**
   - Processed batch/stream data is ultimately loaded into BigQuery for analytics.
   - dbt performs transformations and SQL-based data modeling on top of BigQuery tables for final consumption (analytics, dashboards, or ML feature stores).
5. **Feature Store (Feast)**
   - Tracks feature definitions and materializes features for both online and offline use.
   - Offline store can be your data lake or warehouse (e.g., BigQuery).
   - Online store can be Redis or any NoSQL database, enabling low-latency access to features for real-time inference.
6. **Data Quality & Validation (Great Expectations)**
   - Ensures data integrity during ingestion (batch or streaming).
   - Integrates with Airflow to run checks as part of your ETL/ELT pipelines.
7. **Infrastructure & Deployment**
   - **Docker & Kubernetes (GKE)**: Containerize Pyspark, Flink, Kafka, Airflow, Feast, and other services, then orchestrate them on GKE.

- **CI/CD**: Automate testing, linting, security scans, building/pushing container images, and deploying to Kubernetes.
- **Monitoring & Logging**: Use tools like Prometheus, Grafana, or ELK Stack to observe the health of streaming jobs, batch jobs, and data pipelines.

---

2. Recommended Folder Structure

A typical best-practice folder layout for this kind of project might look like this:

```css
project-root/
├── README.md
├── docker/
│   ├── Dockerfile.airflow
│   ├── Dockerfile.flink
│   ├── Dockerfile.kafkabroker
│   ├── Dockerfile.pyspark
│   ├── Dockerfile.feast
│   └── ...
├── infra/
│   ├── k8s/
│   │   ├── deployments/
│   │   │   ├── airflow-deployment.yaml
│   │   │   ├── flink-deployment.yaml
│   │   │   ├── kafka-deployment.yaml
│   │   │   ├── feast-deployment.yaml
│   │   │   └── ...
│   │   ├── services/
│   │   │   ├── airflow-service.yaml
│   │   │   ├── kafka-service.yaml
│   │   │   └── ...
│   │   └── configs/
│   │       ├── minio-configmap.yaml
│   │       └── ...
│   ├── terraform/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── ...
```

```
│   └── ...
├── configs/
│   ├── airflow/
│   │   └── airflow.cfg
│   ├── spark/
│   │   └── spark-defaults.conf
│   ├── flink/
│   │   └── flink-conf.yaml
│   ├── feast/
│   │   └── feature_store.yaml
│   ├── kafka/
│   │   └── server.properties
│   └── ...
├── scripts/
│   ├── setup_db.sh
│   ├── init_minio.sh
│   ├── mock_event_generation.py
│   └── ...
├── src/
│   ├── batch_ingestion/
│   │   ├── main_batch.py
│   │   ├── transformations.py
│   │   └── ...
│   ├── streaming_ingestion/
│   │   ├── main_stream.py
│   │   ├── flink_jobs/
│   │   │   ├── kafka_consumer.py
│   │   │   └── ...
│   │   ├── pyspark_streaming/
│   │   │   └── ...
│   │   ├── cdc/
│   │   │   ├── debezium_config.json
│   │   │   └── ...
│   │   └── ...
│   ├── feature_store/
│   │   ├── feast/
│   │   │   ├── feature_defs.py
│   │   │   └── registry.db
│   │   └── ...
│   ├── transformations/
│   │   ├── common_utils.py
│   │   ├── aggregator.py
```

```
│   │   └── ...
│   ├── data_validation/
│   │   └── great_expectations/
│   │       ├── expectations/
│   │       ├── checkpoints/
│   │       └── great_expectations.yml
│   └── ...
├── orchestration/
│   └── airflow/
│       ├── dags/
│       │   ├── batch_dag.py
│       │   ├── streaming_dag.py
│       │   └── ...
│       └── plugins/
│           └── ...
├── dbt/
│   ├── dbt_project.yml
│   ├── models/
│   │   ├── staging/
│   │   ├── marts/
│   │   └── ...
│   └── seeds/
├── tests/
│   ├── unit/
│   │   ├── test_batch_ingestion.py
│   │   ├── test_streaming.py
│   │   └── ...
│   ├── integration/
│   │   ├── test_end_to_end.py
│   │   └── ...
│   └── ...
└── .github/workflows/
    ├── ci_cd_pipeline.yml
    └── ...
```

Key Folder Descriptions

1. `docker/` :
   Contains Dockerfiles (one per service). This allows each component (Airflow, Spark, Flink, Feast, Kafka broker, etc.) to be built into separate containers.
2. `infra/` :

- `k8s/`: Kubernetes deployment, service YAML files, and configmaps for GKE deployment.
- `terraform/`: If you use Terraform for provisioning infrastructure (e.g., GCS buckets, BigQuery datasets, GKE clusters, etc.).

3. `configs/`:
   Central location for all configuration files. This might include Spark defaults, Flink configs, Kafka server properties, Feast feature store configuration, etc. Ensures environment-specific configs can be easily referenced and swapped.
4. `scripts/`:
   - Shell and Python scripts for provisioning, local testing, mock data generation, etc.
   - A script to generate and push mock events to a local DB can reside here.
5. `src/batch_ingestion/`:
   - Batch jobs written in PySpark or standard Python scripts that handle ingesting local files into the data lake or BigQuery.
   - Orchestrated by Airflow DAGs to run on a schedule.
6. `src/streaming_ingestion/`:
   - Contains code for streaming architectures with Kafka, Flink, Spark Streaming, etc.
   - Debezium configuration for CDC from the source DB to Kafka topics.
   - Flink or Spark Streaming jobs that read from Kafka, transform data, and write to the data lake or feature store.
7. `src/feature_store/`:
   - Manages feature definitions (Feast) and references to your offline and online stores.
   - Could also contain code for data transformations specific to feature engineering.
8. `src/data_validation/great_expectations/`:
   - Great Expectations configs (checkpoints, expectations, suites) for data validation.
   - Tightly integrated with batch or streaming pipelines (e.g., run validations in Airflow tasks).
9. `orchestration/airflow/`:
   - Your Airflow `dags/` directory that defines all batch and streaming workflows (DAG files).

- `plugins/` for custom Airflow operators, hooks, or macros.

10. `dbt/`:
    - Houses dbt models, seeds, macros for transformations inside BigQuery.
    - `dbt_project.yml` for project configuration.
11. `tests/`:
    - Contains both unit tests (for individual components) and integration tests (for end-to-end data pipeline testing).
12. `.github/workflows/`:
    - GitHub Actions (or other CI/CD pipeline definitions) for automated testing, building Docker images, and deploying to your Kubernetes cluster.

Where to Add Additional Components

- **ML Pipeline Code**: If you have a separate ML pipeline that consumes the curated data or features, you might create an additional top-level directory, e.g., `ml_pipeline/`.
- **Analytics/BI**: If the project includes dashboards or custom reporting tools, you might have a `analytics/` or `reports/` directory to store those assets.

---

3. Summary

- **Batch Pipeline**: Local or scheduled ingestion of files using Pyspark, stored in MinIO/GCS, then loaded/transformed in BigQuery via dbt.
- **Streaming Pipeline**: Mock event generation into a DB, Debezium captures DB changes to Kafka, then Flink/Spark consumes from Kafka, transforms, and writes to data lake or feature store.
- **Feature Store**: Feast for feature definitions, using BigQuery for offline storage and a fast key-value store (e.g., Redis) or minimal-latency DB for serving real-time features.
- **Data Quality & Observability**: Great Expectations integrated with Airflow ensures data is validated at key pipeline steps; logs and metrics are collected via your preferred

monitoring stack.

- **Deployment**: Dockerized everything, orchestrated with Kubernetes (GKE). Terraform or other IaC to manage environment infrastructure. CI/CD pipeline automates tests, builds, and rollouts.

This architecture and folder structure should keep your project maintainable, modular, and conducive to team collaboration. It allows you to configure either local or cloud resources, handle both batch and streaming data, and integrate all the key data-engineering and MLOps tools seamlessly.

What are pyspark and DBT do in my project ? Output DBT is my preprocessed data, but pyspark can handle complex transforms from preprocessed data to become the features ?

2/2

> **Understanding the roles**
>
> I'm digging into the distinct functions of PySpark and DBT in the data pipeline. PySpark handles data engineering and advanced transformations, while DBT focuses on SQL transformations in the data warehouse, specifically using BigQuery for final data storage.
>
> **Clarifying the integration**
>
> I'm piecing together how PySpark tackles complex data transformations for ML, while DBT focuses on SQL-driven transformations in the data warehouse, ensuring a smooth data pipeline.

Here's a quick breakdown of how **PySpark** and **dbt** each fit into your data pipeline and what roles they typically play:

---

1. PySpark

1. **Distributed Batch Processing**
   - PySpark runs on top of a Spark cluster, allowing you to process large datasets in parallel. This is especially useful if you're handling very large batch files, or if you need advanced data transformations and aggregations that can't easily be expressed in SQL alone.

2. **Complex Transformations & Feature Engineering**
    - Spark's Python API (PySpark) offers libraries for machine learning (Spark ML), streaming (Spark Structured Streaming), and advanced transformations (Spark SQL, DataFrame API). This is handy for:
        - **Pre-processing** the raw data from your data lake before loading it into the warehouse.
        - **Feature extraction**, engineering, or scaling that requires code-based logic (e.g., pivoting, custom aggregation, window functions over very large data, or advanced ML transformations).
3. **Multiple Data Sources (Batch + Streaming)**
    - You can connect Spark to a wide variety of data sources: local files, object stores (MinIO/GCS), Kafka (streaming), or any JDBC-based database. This makes PySpark very flexible in your pipeline.

By using PySpark, you can handle the heavy lifting of data wrangling at scale—be it initial cleaning, joining multiple data sets, or creating complex features—before pushing the resulting data to BigQuery or your feature store.

---

2. dbt

1. **SQL-Centric Transformations in the Data Warehouse**
    - dbt ("data build tool") focuses on transformations *inside* the warehouse (BigQuery, in your case).
    - Rather than physically moving data with an external tool, dbt **executes SQL** queries in BigQuery to transform your tables from raw/staging layers into refined, production-ready tables.
    - This approach is sometimes called "ELT" (Extract, Load, then Transform) because you first load the raw data into the warehouse, and then use SQL transformations within the warehouse environment.
2. **Data Modeling & Documentation**

- dbt excels at building clear, version-controlled data models (fact, dimension, staging tables).
- You can layer transformations so that each step is documented, tested (with dbt tests), and fully versioned in Git.
- It's easy to set up incremental loads, snapshots, and macros for reusability.

3. **Collaboration & Testing**
   - Data analysts, engineers, and analytics engineers can collaborate using SQL-based transformations.
   - dbt includes built-in testing (e.g., "check if this column is never null") and documentation generation, which helps ensure data quality in the warehouse.

Overall, dbt is your "last mile" inside BigQuery. It organizes, tests, and maintains well-structured tables that analysts or downstream ML models can query directly.

How They Work Together

1. **Initial Ingestion & Complex Prep (PySpark)**
   - Gather raw data from source files or streaming sources, do heavy transformations, potentially reduce data size, or create specialized aggregates or feature sets that are simpler to load.
   - Write these curated results into staging tables in BigQuery (or store them as intermediate Parquet files in your data lake).
2. **Warehouse Transformation (dbt)**
   - Once the data lands in BigQuery, dbt runs to create refined, analytics-ready (or model-ready) tables.
   - This includes structured data modeling (facts, dimensions) and additional transformations where SQL is more convenient.
3. **Feature Engineering**
   - **Option A**: Some feature engineering is done with PySpark if it involves advanced transformations, machine learning pipelines, or large-scale computational tasks.

- **Option B**: Lighter transformations or feature selection steps can happen in dbt if they're straightforward SQL-based transformations (e.g., filtering, joining with dimension tables).
- **Option C**: You can do both—initial complex transformations in Spark, then simpler final transformations in dbt. Ultimately, you push the final features to your Feature Store (Feast) for model training or real-time serving.

Example Flow

1. **PySpark Batch**
   1. Read raw data from MinIO or GCS.
   2. Clean data (remove duplicates, apply complex transformations).
   3. Write the cleaned or partially transformed data to BigQuery staging tables.
2. **dbt**
   1. Transform staging tables into curated, analytics-ready models using SQL.
   2. Run data quality tests (dbt tests) to confirm expected schemas and data constraints.
   3. Publish final tables (or "marts") for BI or ML consumption.
3. **PySpark/ML**
   1. Optionally, read final data from BigQuery to do advanced ML feature engineering or training in Spark.
   2. Store the resulting feature sets back into the data lake or a Feature Store.
   3. Serve real-time features to your ML applications.

With this design, **PySpark** handles the heavier, code-based data engineering tasks (especially at scale), while **dbt** governs transformations and governance inside BigQuery with best practices for data modeling, version control, and data quality checks.

get me the recommended folder structure after that and describe this

Reasoned for a couple of seconds

Below is a **recommended folder structure** for your project, with **PySpark** and **dbt** clearly separated so that each can do what it does best. Following the structure, you will find a short description of each folder and how it fits into the data pipeline.

Recommended Folder Structure

```css
project-root/
├── README.md
├── docker/
│   ├── Dockerfile.airflow
│   ├── Dockerfile.flink
│   ├── Dockerfile.kafkabroker
│   ├── Dockerfile.pyspark
│   ├── Dockerfile.feast
│   └── ...
├── infra/
│   ├── k8s/
│   │   ├── deployments/
│   │   │   ├── airflow-deployment.yaml
│   │   │   ├── flink-deployment.yaml
│   │   │   ├── kafka-deployment.yaml
│   │   │   ├── feast-deployment.yaml
│   │   │   └── ...
│   │   ├── services/
│   │   │   ├── airflow-service.yaml
│   │   │   ├── kafka-service.yaml
│   │   │   └── ...
│   │   └── configs/
│   │       ├── minio-configmap.yaml
│   │       └── ...
│   ├── terraform/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── ...
│   └── ...
├── configs/
│   ├── airflow/
```

```
│   │   └── airflow.cfg
│   ├── spark/
│   │   └── spark-defaults.conf
│   ├── flink/
│   │   └── flink-conf.yaml
│   ├── feast/
│   │   └── feature_store.yaml
│   ├── kafka/
│   │   └── server.properties
│   └── ...
├── scripts/
│   ├── setup_db.sh
│   ├── init_minio.sh
│   ├── mock_event_generation.py
│   └── ...
├── src/
│   ├── pyspark_jobs/
│   │   ├── batch_ingestion/
│   │   │   ├── main_batch.py
│   │   │   ├── transformations.py
│   │   │   └── ...
│   │   ├── streaming_ingestion/
│   │   │   ├── main_stream.py
│   │   │   ├── flink_jobs/
│   │   │   │   ├── kafka_consumer.py
│   │   │   │   └── ...
│   │   │   ├── pyspark_streaming/
│   │   │   │   └── ...
│   │   │   ├── cdc/
│   │   │   │   ├── debezium_config.json
│   │   │   │   └── ...
│   │   │   └── ...
│   │   ├── feature_engineering/
│   │   │   ├── feature_calc.py
│   │   │   └── ...
│   │   └── utils/
│   │       ├── common_utils.py
│   │       └── ...
│   ├── data_validation/
│   │   └── great_expectations/
│   │       ├── expectations/
│   │       ├── checkpoints/
```

```
│   │       └── great_expectations.yml
│   └── feature_store/
│       ├── feast/
│       │   ├── feature_defs.py
│       │   └── registry.db
│       └── ...
├── orchestration/
│   └── airflow/
│       ├── dags/
│       │   ├── batch_dag.py
│       │   ├── streaming_dag.py
│       │   └── ...
│       └── plugins/
│           └── ...
├── dbt/
│   ├── dbt_project.yml
│   ├── models/
│   │   ├── staging/
│   │   ├── marts/
│   │   └── ...
│   └── seeds/
├── tests/
│   ├── unit/
│   │   ├── test_batch_ingestion.py
│   │   ├── test_streaming.py
│   │   └── ...
│   ├── integration/
│   │   ├── test_end_to_end.py
│   │   └── ...
│   └── ...
└── .github/workflows/
    ├── ci_cd_pipeline.yml
    └── ...
```

Folder-by-Folder Description

1. `README.md`

- Explains your project's purpose, setup instructions, and any noteworthy details.

2. `docker/`

- **Dockerfiles** for each service (Airflow, Flink, Kafka, PySpark, Feast, etc.). This ensures each component can be containerized for easy deployment to Kubernetes or local Docker.

3. `infra/`

- `k8s/`: All Kubernetes YAML files for deploying pods, services, and config maps.
- `terraform/`: If you use Terraform to provision cloud resources (like GCS buckets, BigQuery datasets, GKE clusters), these files live here.

4. `configs/`

- Houses all **configuration files** for tools and services—Spark configs, Flink configs, Kafka properties, Feast feature store config, etc. This allows you to manage environment-specific settings in one place.

5. `scripts/`

- **Shell/Python scripts** for environment setup, database initialization, or generating mock data (e.g., `mock_event_generation.py` to produce test events for streaming pipelines).

6. `src/`

- **Main application code** that handles data ingestion, transformations, and feature engineering.
  - `pyspark_jobs/`:
    - `batch_ingestion/`: PySpark jobs to process local files in batch mode. Typically does the heavy-lifting data wrangling/cleaning before loading into the data lake or BigQuery.
    - `streaming_ingestion/`: PySpark (or Flink/Spark streaming) jobs for real-time data flows from Kafka, Debezium, etc.
    - `feature_engineering/`: Code specifically focused on deriving advanced features or performing ML pre-processing transformations at scale.
    - `utils/`: Utility scripts or helper functions used across PySpark jobs.
  - `data_validation/great_expectations/`: Contains your Great Expectations config (`great_expectations.yml`), expectations, checkpoints, and more, used to validate data at key points in the pipeline.

- `feature_store/feast/`: If you're using Feast for feature management, store feature definitions (e.g., `feature_defs.py`) and local registry files here.

7. `orchestration/airflow/`

- `dags/`: Where your Airflow DAGs reside—e.g., `batch_dag.py` that orchestrates PySpark batch runs, `streaming_dag.py` for continuous or scheduled streaming jobs, etc.
- `plugins/`: Custom Airflow operators, hooks, or macros that you might need.

8. `dbt/`

- **SQL-based transformations** inside BigQuery.
- `dbt_project.yml` sets up your project.
- `models/` subdirectories (e.g., `staging/`, `marts/`) hold `.sql` files that define transformations for staging, fact, and dimension tables.
- `seeds/` for static CSV data seeds that can be loaded into the warehouse as reference tables.

9. `tests/`

- `unit/`: Tests for individual PySpark transformations, streaming processes, or utility functions.
- `integration/`: End-to-end pipeline tests. For example, verifying that data flows from mock events through PySpark, to BigQuery, then is transformed by dbt as expected.

10. `.github/workflows/`

- **CI/CD pipeline definitions** (GitHub Actions, for example) that run tests, build Docker images, and deploy to Kubernetes.

Roles of PySpark and dbt

1. **PySpark**
    - **High-Scale Data Wrangling**: Ideal for dealing with very large datasets, advanced transformations, or combining multiple data sources in a code-based (Python) manner.

- **Complex Feature Engineering**: Use PySpark's DataFrame and Spark ML libraries for large-scale feature generation, e.g., window aggregations, pivoting, custom logic.
- **Batch + Streaming**: PySpark can handle both scheduled batch jobs (e.g., reading large parquet files) and streaming ingestion (e.g., reading from Kafka or Debezium topics in near real time).

2. **dbt**
   - **SQL-Driven Transformations**: Executes transformations directly in BigQuery using SQL. Perfect for building clean staging tables and refined data "marts" (fact/dim tables).
   - **Data Modeling & Testing**: Provides version-controlled SQL models, built-in tests, and documentation generation.
   - **Incremental & Collaborative**: Teams of data analysts/engineers can iteratively develop transformations, automatically run tests, and keep transformations in sync via Git.

In a typical flow, PySpark might handle **initial** heavy transformations on raw data (removing duplicates, merging datasets, advanced feature engineering) before loading curated data into BigQuery. Then **dbt** takes over, structuring and refining that data within BigQuery using straightforward, maintainable SQL-based transformations.

Putting It All Together

- **Ingestion**: Data enters either via batch files (processed by PySpark) or streaming events (picked up by Spark/Flink from Kafka + Debezium).
- **Landing**: The cleaned or partially transformed data is stored in a data lake (MinIO / GCS) or loaded into BigQuery staging tables.
- **Transformation & Modeling**: dbt runs inside BigQuery to finalize transformations, create well-modeled tables, and validate data quality with built-in tests.
- **Feature Store**: If you need advanced features for machine learning, you can store them in Feast, which references either BigQuery or a fast online store like Redis.
- **Orchestration**: Airflow coordinates everything: triggers PySpark jobs, runs dbt transformations, executes Great Expectations checks, and schedules streaming or batch

pipelines.

By splitting responsibilities this way and organizing your files in a clear folder structure, the project remains scalable, maintainable, and easier to collaborate on across data engineering, analytics, and ML teams.

should i put dbt in src folder

Reasoned for a couple of seconds

In most cases, **it's best to keep your dbt project in its own top-level directory**, rather than bundling it under `src/`. The reason is that dbt is more than just code—it's also a framework for modeling, testing, and documenting your data within the warehouse. Storing dbt at the top level helps keep your SQL-based modeling and transformations logically separate from application code (e.g., PySpark scripts) and makes it easier for data analysts or analytics engineers to work directly with dbt without sifting through code directories.

Why keep dbt outside of `src/`?

1. **Clarity of Purpose**
    - `src/` typically houses Python or application-layer code—like PySpark jobs, streaming tasks, or utilities for data ingestion.
    - dbt transformations, on the other hand, are primarily SQL-based, with a metadata/configuration layer (dbt_project.yml, seeds, macros, etc.). Keeping them separate clarifies which transformations are code-based (in `src/`) vs. which are SQL-based within the warehouse (dbt).
2. **Separate Lifecycle & Tooling**
    - dbt has its own commands and workflows (e.g. `dbt run`, `dbt test`, `dbt docs generate`), often tied to analytics/BI workflows.
    - PySpark jobs are orchestrated differently (e.g. run on Spark clusters, triggered by Airflow tasks).
    - Having a dedicated `dbt/` folder makes it simpler to manage dbt as an independent module or package with its own CI/CD steps and versioning.
3. **Community Conventions**
    - The broader dbt community and documentation typically assume a top-level `dbt/` folder. This aligns your repo structure with standard dbt best practices and makes it more discoverable for new contributors.

Example

A common project layout is:

```css
project-root/
├── src/
│   ├── pyspark_jobs/
│   ├── data_validation/
│   └── ...
├── dbt/
│   ├── dbt_project.yml
│   ├── models/
│   ├── seeds/
│   └── ...
├── orchestration/
└── ...
```

This way, the **dbt** directory stands alone, is easy to locate, and is independent from your Python application code in `src/`.

give me a system architecture for detail then describe it

> **Clarifying system elements**
>
> Crafting a straightforward system architecture dashboard with PySpark, Flink, Kafka, Airflow, Feast, Docker, K8S, CICD, DBT, Great Expectations, a Data Warehouse in BigQuery, and a data lake in MinIO or GCS.
>
> **Mapping the pipeline**
>
> I'm outlining the data flow: Point data ingestion, highlight orchestration, quarantine data validation, and guide towards final consumption. Progress is being made step by step.

Below is a **detailed system architecture** that stitches together all of the components you mentioned (PySpark, Flink, Kafka, Airflow, Feast, Docker, Kubernetes, CI/CD, dbt, Great Expectations, MinIO/GCS, BigQuery). After the schematic view, you'll find a section-by-section explanation describing how each piece works together in the pipeline.

System Architecture Diagram (Conceptual Flow)

```pgsql
 ┌───────────────────┐        ┌──────────────────────┐
 │   Local Batch     │        │      Source DB       │
 │   Files           │        │ (Mock Events Insert) │
 └─────────┬─────────┘        └──────────┬───────────┘
           │                             │
           ▼                             ▼
 ┌───────────────────┐        ┌──────────────────────┐
 │   PySpark Batch   │        │       Debezium       │
 │(read files, clean,│        │ (CDC connector)      │
 │ transform)        │        └──────────┬───────────┘
 └─────────┬─────────┘                   │
           │                             │
           ▼                             ▼
 ┌────────────────────────────┐     ┌──────────────────────────┐
 │   Data Lake Storage        │←────┤          Kafka           │
 │ (MinIO or GCS)             │     │ (Debezium → Kafka Topic) │
 └─────────┬──────────────────┘     └──────────┬───────────────┘
           │                                   │
           ▼                                   ▼
 ┌────────────────────────┐        ┌────────────────────────┐
 │  Airflow (Orchestration)        │     Flink / Spark      │
 │  - schedules jobs      │        │     Streaming          │
 │  - triggers validations│        │  (reads from Kafka,    │
 │  - runs dbt and CI tasks        │   transforms data)     │
 └────────────────────────┘        └───────────┬────────────┘
                                               │
                                               ▼
                            ┌─────────────────────────────────┐
                            │   Data Validation (GE)          │
                            │(Great Expectations checks)      │
                            └──────────┬──────────────────────┘
                                       │
                                       ▼
                            ┌─────────────────────────────────┐
                            │   Data Warehouse (BigQuery)     │
                            │  (staging + curated tables)     │
                            └──────────┬──────────────────────┘
                                       │
```

```
                    ▼
          ┌────────────────────────────────┐
          │    dbt (SQL Transformations)   │
          │ (models, seeds, tests, docs)   │
          └────────────────────────────────┘
                    │
                    │
                    ▼
          ┌────────────────────────────────┐
          │   Feature Store (Feast)        │
          │ (Offline store in BQ, optional │
          │  online store in Redis, etc.)  │
          └────────────────────────────────┘
                    │
                    │
                    ▼
          ┌────────────────────────────────┐
          │   ML Modeling / Consumption    │
          │  (e.g., Notebooks, Online Inference,
          │   BI, or downstream apps)      │
          └────────────────────────────────┘
```

Architecture Explanation

1. **Data Sources**
   - **Local Batch Files**: Periodically, raw data files (CSV, JSON, or Parquet) are placed in a local folder or an on-prem data store.
   - **Transactional DB (Mock Events):** A relational database (e.g., PostgreSQL or MySQL) that receives new events (inserts/updates/deletes). These events represent real-time data that will be captured by **Debezium** for streaming pipelines.
2. **Change Data Capture with Debezium + Kafka**
   - **Debezium** continuously monitors the transactional DB's write-ahead log (WAL) and publishes change events to **Kafka** topics in real time.
   - This setup is crucial for capturing incremental updates, enabling streaming ingestion.
3. **Batch Ingestion with PySpark**

- **PySpark** jobs ingest large batch files from local storage. These jobs:
    - Clean and preprocess the data.
    - Potentially do more advanced transformations (joins, aggregations) that are easier in a distributed computing environment.
- The **output** can be stored in your **Data Lake** (MinIO or Google Cloud Storage) or loaded directly into **BigQuery** as a staging table.

4. **Data Lake (MinIO / GCS)**
    - Either on-prem (MinIO) or cloud-based (Google Cloud Storage).
    - Serves as a landing zone for raw or partially processed data.
    - Both the batch jobs (PySpark output) and streaming jobs (Flink or Spark streaming) can write here.
5. **Flink or Spark Streaming (Real-Time Pipeline)**
    - **Flink** (or Spark Structured Streaming) subscribes to **Kafka** topics generated by Debezium or other producers.
    - Processes these incoming events in near real time: filtering, aggregations, or transformations.
    - Writes the real-time processed data back to your data lake or **Feature Store** (Feast), or directly into **BigQuery** if needed.
6. **Airflow (Orchestration)**
    - Coordinates all pipelines (batch or streaming).
    - Schedules PySpark batch jobs (e.g., daily or hourly).
    - Triggers dbt runs after new data arrives in BigQuery.
    - Integrates with **Great Expectations** for data validation steps.
7. **Data Validation (Great Expectations)**
    - **Great Expectations** runs within the pipeline to validate data at key checkpoints:
        - After ingestion from PySpark batch or streaming.
        - Before or after loading data into BigQuery.
    - Ensures your data meets defined constraints (no nulls in critical fields, correct data ranges, etc.).
8. **Data Warehouse (BigQuery)**

- Houses staging and curated tables.
- Long-term, you might keep raw or lightly processed data in one schema (staging), then use transformations to produce analytics-ready tables in another (marts, production).

9. **dbt (SQL Transformations in BigQuery)**
   - Executes transformations *inside* BigQuery using SQL.
   - Creates data models (facts, dimensions) for analytics and ML consumption.
   - Maintains version-controlled transformations, built-in testing, and documentation.
10. **Feature Store (Feast)**

- Stores feature definitions and metadata.
- **Offline Store** often uses BigQuery (for training data creation).
- **Online Store** can be a low-latency DB (e.g., Redis) used to serve features in real-time inference pipelines.

11. **ML Modeling & Consumption**

- Data scientists can pull curated or feature-engineered data from BigQuery or Feast to train ML models.
- Deployed models can also fetch features in real time from Feast's online store.

12. **Infrastructure & Deployment**

- **Docker**: Containerize each service (PySpark, Flink, Kafka, Airflow, Feast, dbt, etc.).
- **Kubernetes (GKE)**: Orchestrate and scale containers in Google Kubernetes Engine (or any other Kubernetes environment).
- **CI/CD**: Automates building, testing, and deploying containers. Integrates with version control (GitHub/GitLab) to trigger pipelines.

13. **Monitoring & Observability**

- Tools such as Prometheus, Grafana, and the ELK stack (Elasticsearch, Logstash, Kibana) often plug into each component (Airflow, Kafka, Flink, Spark) to provide logs, metrics, alerts, and dashboards.

Key Takeaways

1. **Batch & Streaming**: You can handle both periodic large-file ingestions with PySpark **and** real-time event streaming with Debezium + Kafka → Flink.
2. **Data Lake + Warehouse**: Raw or partially processed data lands in MinIO/GCS (the data lake), while structured, analytics-ready data lives in BigQuery (the data warehouse).
3. **dbt**: Transforms and models your data in a maintainable, SQL-centric workflow.
4. **Great Expectations**: Adds data quality checks at each stage of the pipeline.
5. **Feature Store**: Feast manages and serves your ML features, bridging offline (BigQuery) and online (Redis or other) stores.
6. **Orchestration & CI/CD**: Airflow coordinates the data pipeline steps, and an automated CI/CD pipeline handles testing and deployment on Kubernetes.

With this architecture, your system can ingest data from multiple sources (batch or streaming), store it flexibly (on-prem or cloud), validate its quality, transform it for analytics or ML, and serve it efficiently to production applications.

should DBT handle the steaming data workflow ? or just only in batch process

> **Crafting the context**
>
> dbt is traditionally used for batch transformations in a data warehouse, usually with micro-batching and incremental models.
>
> **Highlighting incremental utility**
>
> dbt can handle near real-time transformations with incremental models, but it's not suited for high-volume real-time streaming, which is better managed by frameworks like Flink, Spark, or Kafka.

Short Answer:
dbt is primarily a **batch-oriented** transformation tool that operates inside the data warehouse (BigQuery). It's *not* designed to directly process streaming data in real time. Instead, you generally rely on a streaming framework (e.g., Flink, Spark Structured Streaming) to capture and process events before they land in the warehouse. Once the data is in BigQuery—even if it arrives frequently—dbt can run in **micro-batches** or on a schedule to transform newly ingested data.

Detailed Explanation

1. **dbt's Core Purpose**
   - dbt ("data build tool") is built around **SQL transformations** that execute directly in the warehouse (BigQuery, Snowflake, etc.).
   - It is designed for version-controlled data modeling, testing, and documentation, following a batch or micro-batch (incremental) pattern.
2. **Handling Real-Time Data**
   - In a streaming use case, data flows continuously through tools like **Debezium → Kafka → Flink** (or Spark Streaming).
   - That pipeline *does* the heavy lifting of capturing real-time events, transforming them on the fly, and loading them into a data store—often the data lake (MinIO, GCS) and/or staging tables in BigQuery.
   - dbt then picks up the data **once it's in BigQuery**. You can run dbt in short intervals (e.g., every few minutes) to "catch up" with new data, or schedule it to run after streaming loads complete.
3. **Incremental Models**
   - dbt supports **incremental models**, which let you process only the new/updated data since the last run.
   - This can approximate a near-real-time pattern if you schedule dbt to run frequently. However, it's still not an event-driven streaming architecture—dbt waits for the data to be present in BigQuery before it transforms it.
4. **Where Streaming Logic Lives**
   - True streaming logic (windowing, event-time operations, out-of-order data handling) typically lives in **Flink** or **Spark Streaming** jobs that read from Kafka.
   - Once those frameworks have processed data, you persist it in BigQuery or your data lake. At that point, dbt can do final SQL transformations, metrics calculations, or data modeling steps.
5. **Best Practice**
   - Let **Flink/Spark** handle the real-time ingestion and initial transformations.

- Once data arrives in BigQuery—be it every few seconds, minutes, or hours—**dbt** can then transform the new arrivals into refined tables.
- If the transformations are relatively simple (SQL-based filtering, joining, etc.), you can rely more on dbt. If you need complex transformations (e.g., real-time feature engineering), do it upstream in Flink/Spark first.

Example: Streaming + dbt Flow

1. **Flink/Spark** ingests data from Kafka in near real time.
2. The transformed or raw data is loaded into **BigQuery** staging tables.
3. An **Airflow** DAG triggers **dbt** on a schedule or upon successful arrival of streaming batches.
4. **dbt** performs incremental transformations in BigQuery, updating downstream fact/dimension tables.
5. Data is now “analytics ready” in near real time, or as frequently as your dbt job runs.

In summary, dbt is excellent for version-controlled, testable SQL transformations inside the warehouse, but it is not a streaming tool. If you have real-time pipelines, pair dbt with Flink/Spark for the streaming leg, then run dbt in frequent intervals for the transformations once data lands in BigQuery.

the features table with complexity logic of transformation, that be used in AIML model, what is the solution handle it ? and how ? How can process handle the batch data and streaming data when there was the new one come to ?

Below is an explanation and solution design for handling a features table—especially when it contains complex transformation logic and must serve an AI/ML model—by unifying batch and streaming pipelines.

Overview

When your AI/ML model relies on a features table that requires complex transformations, you need to ensure that both historical (batch) and new (streaming) data are processed consistently. The goal is to update your feature set in real time while also allowing for comprehensive recalculations as more data accumulates.

Solution Components

1. **Feature Store (Feast)**
    - **Unified Serving Layer:**
      Feast provides an abstraction for both offline (training) and online (inference) features. It enables you to maintain a single source of truth for your features, ensuring consistency regardless of whether the data comes from batch or streaming pipelines.
2. **Batch Processing (PySpark + dbt)**
    - **Heavy Transformations:**
      Use PySpark to process large volumes of historical data. This includes complex transformation logic such as aggregations, window functions, and feature engineering that require distributed processing.
    - **Data Modeling:**
      Once the data is cleansed and processed in PySpark, it can be loaded into BigQuery. dbt then takes over for SQL-based transformation, data modeling, and incremental updates of the features table.
3. **Streaming Processing (Flink/Spark Streaming)**
    - **Real-Time Updates:**
      A streaming framework (Flink or Spark Structured Streaming) continuously processes incoming events—such as those captured from Debezium and published to Kafka—to update features on the fly.
    - **Micro-Batching & Incremental Logic:**
      Even though streaming jobs handle near real-time events, they can use micro-

batching techniques to process small chunks of data. This ensures that the transformation logic is applied incrementally and consistently.

4. **Orchestration (Airflow)**
   - **Scheduling & Coordination:**
     Airflow orchestrates both batch and streaming workflows. For batch jobs, it schedules periodic PySpark runs and triggers dbt jobs to refresh the features table in BigQuery.
     For streaming, Airflow may monitor the pipeline health and coordinate fallback or reprocessing strategies if needed.

How It All Comes Together

1. **Unified Transformation Logic:**
   - **Shared Code Base:**
     To ensure consistency, you should implement the core transformation logic as a shared library. Both your PySpark batch jobs and your streaming jobs can import this library so that features are computed in the same way, regardless of data arrival mode.
2. **Batch Workflow:**
   - **Ingestion & Transformation:**
     - PySpark reads historical data from the data lake (MinIO/GCS).
     - It applies complex transformation logic (feature engineering) on the large dataset.
     - The resulting data is written into staging tables in BigQuery.
   - **Final Modeling with dbt:**
     - dbt transforms these staging tables into the final features table, applying incremental models to refresh only new or updated data.
   - **Integration with Feast:**
     - The curated features are then exported to Feast’s offline store for training and its online store for real-time serving.

3. **Streaming Workflow:**
   - **Real-Time Event Handling:**
     - Streaming engines (Flink/Spark Streaming) continuously read new events from Kafka.
     - They apply the same transformation logic (via the shared code) on micro-batches or continuous streams.
     - The processed real-time data is immediately ingested into an online store or a staging area in BigQuery.
   - **Near Real-Time Updates:**
     - Airflow can trigger frequent dbt runs (or an incremental refresh mechanism) to ensure that streaming updates are merged into the overall features table.
     - Feast then uses these updated features for real-time inference.
4. **Handling New Data:**
   - **Incremental Updates:**
     Both workflows should be designed to handle incremental updates. For batch jobs, this means processing only the delta (new data) during each run. For streaming jobs, micro-batching ensures that as soon as new events come in, the features are updated.
   - **Consistency & Overlap:**
     The feature store abstracts the underlying dual pipelines, so whether the feature was updated via batch or streaming, the AI/ML model always accesses a consistent view of the features.

Implementation Steps

1. **Develop Shared Transformation Modules:**
   Create a common library (e.g., in Python) that defines your complex feature transformation logic. Both batch (PySpark) and streaming (Flink/Spark Streaming) pipelines import this module.
2. **Set Up Batch Processing:**

- Use PySpark to process historical data.
- Load processed data into BigQuery staging tables.
- Schedule dbt jobs (possibly with Airflow) to transform staging data into the final features table.

3. **Set Up Streaming Processing:**
   - Configure Debezium to capture DB changes and publish events to Kafka.
   - Develop a streaming job using Flink or Spark Streaming that reads from Kafka and applies the shared transformation logic.
   - Load or merge these real-time updates into BigQuery or directly into Feast's online store.
4. **Orchestrate with Airflow:**
   - Create Airflow DAGs to run batch jobs on a schedule.
   - Optionally, monitor streaming job health and trigger dbt jobs frequently (e.g., every few minutes) to capture incremental changes.
5. **Integrate with Feast:**
   - Configure Feast to pull feature data from BigQuery (offline) and update its online store.
   - Ensure that Feast's feature definitions match the transformation logic implemented in your pipelines.

Summary

- **Batch Processing:**
  Heavy transformations are handled by PySpark and refined in BigQuery with dbt. This ensures reliable, reproducible feature computation over large datasets.
- **Streaming Processing:**
  Real-time events are captured by a streaming engine (Flink/Spark Streaming) and processed using the same transformation logic. This pipeline updates the online store for immediate inference.

- **Unified Feature Serving:**
  A feature store like Feast unifies both data flows, ensuring that your AI/ML model always has access to up-to-date, consistent features.

By combining these pipelines, you achieve a robust, scalable solution that updates your features table from both historical and new data streams, keeping your AI/ML models effective and current.

So, The DBT is handle the staging data (both batch and streamming) that store in bigquery, transform it to feature tables, but there is several tasks on DBT belong to this data attributes such as batch or streaming data, right ?

That's correct. dbt sits on top of BigQuery where both batch and streaming data land into staging tables. It doesn't care about the ingestion method—instead, it focuses on transforming that data into clean, analytics-ready feature tables. You can design dbt tasks to handle incremental updates (ideal for streaming) as well as full-refresh models (often used for batch data). This way, you can tailor transformations according to the specific attributes or requirements of batch versus streaming data while maintaining a single, unified source for your AI/ML features.

Give me the final architecture, the final folder structure and describe each component and the data flow

Below is a comprehensive final design that brings together all the components—data ingestion (batch and streaming), transformation, and feature serving—along with a detailed folder structure and a description of each component and data flow.

Final System Architecture

```plaintext
                     ┌─────────────────────────┐
                     │      Data Sources       │
                     │  ┌───────────────────┐  │
                     │  │ Local Batch Files │  │
                     │  └───────────────────┘  │
```

```
                     │  ┌─────────────────────┐  │
                     │  │ Transactional DB    │  │
                     │  │ (for CDC events)    │  │
                     │  └─────────────────────┘  │
                     └────────────┬──────────────┘
                                  │
                                  │
                 ┌────────────────┼──────────────────────┐
                 │                                       │
                 ▼                                       ▼
        ┌─────────────────────┐                ┌─────────────────────┐
        │   PySpark Batch     │                │     Debezium        │
        │   Ingestion &       │                │ (CDC from DB to     │
        │   Transformations   │                │  Kafka)             │
        └─────────┬───────────┘                └──────────┬──────────┘
                  │                                       │
                  │                                       ▼
                  │                               ┌───────────────────┐
                  │                               │      Kafka        │
                  │                               │ (Event Streaming) │
                  ▼                               └──────────┬────────┘
        ┌─────────────────────┐                              │
        │ Data Lake Storage   │  (MinIO or GCS)              │
        │ (Landing zone for   │◄─────────────────────────────┘
        │  raw/processed      │
        │  data from both     │
        │  pipelines)         │
        └─────────┬───────────┘
                  │
                  ▼
            ┌───────────────────┐
            │  BigQuery         │
            │ (Staging Tables)  │
            └─────────┬─────────┘
                  │
                  ▼
            ┌───────────────────┐
            │       dbt         │
            │  (SQL Transform-  │
            │   ations to build │
            │   feature tables) │
            └─────────┬─────────┘
                  │
```

```
          ▼
┌─────────────────┐
│  Feature Store  │
│     (Feast)     │
└────────────┬────┘
          │
          ▼
┌─────────────────┐
│ ML Models /     │
│ Downstream Apps │
└─────────────────┘
```

Key Points of the Architecture

1. **Data Sources**
    - **Local Batch Files:** Files (CSV, Parquet, etc.) are ingested by PySpark for historical or bulk data processing.
    - **Transactional Database:** Captures events in real time (e.g., inserts, updates) that are picked up via CDC using Debezium.
2. **Ingestion Pipelines**
    - **Batch Processing with PySpark:**
        - Reads large files from local storage or object stores.
        - Applies complex, distributed transformations and feature engineering.
        - Outputs the processed data to the Data Lake.
    - **Streaming Processing with Debezium & Kafka:**
        - Debezium monitors the transactional DB and publishes change events to Kafka.
        - A streaming engine (Flink or Spark Streaming) consumes these events, applies incremental transformations, and writes results into the Data Lake or directly into BigQuery staging.
3. **Data Lake (MinIO / GCS)**
    - Serves as a unified landing zone for both batch and streaming outputs.
    - Provides flexibility to store raw and partially processed data before loading it into BigQuery.
4. **Data Warehouse (BigQuery)**

- Acts as the staging area where both batch and streaming data land.
- Supports incremental loading so that new data is continuously integrated.

5. **Transformations with dbt**
   - dbt takes the staging tables in BigQuery and runs SQL-based transformations to build clean, curated feature tables.
   - Supports incremental models to process both full-batch updates and micro-batches from streaming pipelines.
   - The final feature tables are then used for analytics and feeding ML models.
6. **Feature Store (Feast)**
   - Unifies offline (training) and online (inference) features.
   - Pulls curated features from BigQuery and serves them via an online store (e.g., Redis) for real-time scoring.
7. **Downstream Consumption**
   - ML models, BI tools, or any downstream application consume the final feature tables and/or real-time features from Feast.
8. **Orchestration and Monitoring**
   - **Airflow:** Orchestrates both batch and streaming jobs. Schedules PySpark and dbt runs, and monitors overall pipeline health.
   - **CI/CD, Docker, Kubernetes (GKE):** Containerize and deploy components, ensuring smooth operations and scalability.
   - **Great Expectations:** Embedded at key points to validate data quality.

Final Folder Structure

```css
project-root/
├── README.md
├── docker/
│   ├── Dockerfile.airflow
│   ├── Dockerfile.flink
```

```
│   ├── Dockerfile.kafka
│   ├── Dockerfile.pyspark
│   ├── Dockerfile.feast
│   └── ...
├── infra/
│   ├── k8s/
│   │   ├── deployments/
│   │   │   ├── airflow-deployment.yaml
│   │   │   ├── flink-deployment.yaml
│   │   │   ├── kafka-deployment.yaml
│   │   │   ├── feast-deployment.yaml
│   │   │   └── ...
│   │   ├── services/
│   │   │   ├── airflow-service.yaml
│   │   │   ├── kafka-service.yaml
│   │   │   └── ...
│   │   └── configs/
│   │       ├── minio-configmap.yaml
│   │       └── ...
│   ├── terraform/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── ...
│   └── ...
├── configs/
│   ├── airflow/
│   │   └── airflow.cfg
│   ├── spark/
│   │   └── spark-defaults.conf
│   ├── flink/
│   │   └── flink-conf.yaml
│   ├── feast/
│   │   └── feature_store.yaml
│   ├── kafka/
│   │   └── server.properties
│   └── ...
├── scripts/
│   ├── setup_db.sh
│   ├── init_minio.sh
│   ├── mock_event_generation.py
│   └── ...
├── src/
```

```
│   ├── pyspark_jobs/
│   │   ├── batch_ingestion/
│   │   │   ├── main_batch.py
│   │   │   ├── transformations.py
│   │   │   └── ...
│   │   ├── streaming_ingestion/
│   │   │   ├── main_stream.py
│   │   │   ├── flink_jobs/
│   │   │   │   ├── kafka_consumer.py
│   │   │   │   └── ...
│   │   │   ├── pyspark_streaming/
│   │   │   │   └── ...
│   │   │   ├── cdc/
│   │   │   │   ├── debezium_config.json
│   │   │   │   └── ...
│   │   │   └── ...
│   │   ├── feature_engineering/
│   │   │   ├── feature_calc.py
│   │   │   └── ...
│   │   └── utils/
│   │       ├── common_utils.py
│   │       └── ...
│   ├── data_validation/
│   │   └── great_expectations/
│   │       ├── expectations/
│   │       ├── checkpoints/
│   │       └── great_expectations.yml
│   └── feature_store/
│       ├── feast/
│       │   ├── feature_defs.py
│       │   └── registry.db
│       └── ...
├── orchestration/
│   └── airflow/
│       ├── dags/
│       │   ├── batch_dag.py
│       │   ├── streaming_dag.py
│       │   └── ...
│       └── plugins/
│           └── ...
├── dbt/
│   ├── dbt_project.yml
```

```
│   ├── models/
│   │   ├── staging/
│   │   ├── marts/
│   │   └── ...
│   └── seeds/
├── tests/
│   ├── unit/
│   │   ├── test_batch_ingestion.py
│   │   ├── test_streaming.py
│   │   └── ...
│   ├── integration/
│   │   ├── test_end_to_end.py
│   │   └── ...
│   └── ...
└── .github/workflows/
    ├── ci_cd_pipeline.yml
    └── ...
```

Component Descriptions & Data Flow

1. **Data Sources**
   - **Local Batch Files:**
     Files containing historical or bulk data are placed in a local directory (or object storage) and ingested by PySpark.
   - **Transactional DB:**
     Real-time events are written to a relational database. Debezium monitors the DB's change log and publishes events to Kafka.
2. **Ingestion Pipelines**
   - **PySpark Batch Jobs:**
     These jobs read local batch files, apply complex transformations and feature engineering, and write the processed results to the Data Lake.
   - **Streaming Jobs:**
     Debezium captures changes from the DB and pushes them to Kafka. A streaming engine (Flink or Spark Streaming) consumes these events, applies incremental

transformations (using shared transformation logic), and outputs the processed data to the Data Lake or directly into BigQuery staging tables.

3. **Data Lake (MinIO / GCS)**
   - Acts as a landing zone for both pipelines, storing raw or lightly transformed data. This ensures that all data—whether batch or streaming—can later be integrated into the data warehouse.
4. **Data Warehouse (BigQuery)**
   - The processed data from the Data Lake is loaded into BigQuery staging tables. This staging area serves as the input for further transformations.
5. **Transformation with dbt**
   - **dbt** runs SQL-based transformation workflows over the staging tables in BigQuery.
   - It produces well-modeled, version-controlled feature tables by applying incremental (or full-refresh) logic that can handle both batch and streaming updates.
   - The output is a unified, analytics-ready feature set that feeds the Feature Store.
6. **Feature Store (Feast)**
   - Feast serves as the centralized feature repository.
   - It pulls data from BigQuery (offline store) and can update an online store for low-latency access during model inference.
7. **Downstream Consumption**
   - AI/ML models and other applications retrieve features either directly from BigQuery (for training and batch inference) or from Feast's online store (for real-time scoring).
8. **Orchestration & Monitoring**
   - **Airflow** schedules and monitors both the PySpark and dbt jobs as well as any streaming job health checks.
   - **Docker, Kubernetes (GKE), and CI/CD Pipelines** ensure containerized deployment, scaling, and automated testing.
   - **Great Expectations** is integrated at key points in the pipeline to verify data quality.

Summary

- **Unified Data Ingestion:**
  Batch files and streaming events are ingested separately using PySpark (batch) and Debezium–Kafka–Flink/Spark Streaming (real time), and both deposit data into a common Data Lake.
- **Centralized Transformation:**
  Data is loaded into BigQuery staging tables, where dbt applies consistent, SQL-based transformations—whether data arrived in bulk or incrementally—to create refined feature tables.
- **Feature Serving:**
  The final features are exposed through Feast, enabling both offline ML training and real-time inference.
- **Clear Folder Structure & Orchestration:**
  The repository is organized into distinct areas for infrastructure, configuration, source code (for batch/streaming), data validation, orchestration (Airflow), and dbt. This separation simplifies collaboration and maintenance while ensuring each component's role is clear in the overall data flow.

This final architecture and folder layout ensure your system is scalable, maintainable, and capable of handling both historical batch and real-time streaming data to produce consistent features for your AI/ML models.

in the architecture, I dont see Flink (or similar solution), greate expectation, airflow ?

which place the data steamming after consume from kafka should store on datalake or staging data warehouse ?

In our design, Flink (or an equivalent streaming engine), Great Expectations, and Airflow are all part of the overall pipeline—even if they aren't separately drawn as boxes in every diagram. Here's how they fit in and where the streaming data should land:

Role of the Components

- **Flink (or Similar Streaming Engine):**

    - **Purpose:** Consumes real-time events from Kafka (which receives CDC events from Debezium) and applies incremental or micro-batch transformations.
    - **Output:** Processes streaming data in near real time.
- **Great Expectations:**
    - **Purpose:** Validates data quality at key checkpoints in your pipeline. You can run tests after batch jobs or after streaming data is processed to ensure that the data meets defined quality standards.
- **Airflow:**
    - **Purpose:** Orchestrates your entire pipeline. It schedules and monitors both the batch (PySpark + dbt) and streaming jobs (Flink/Spark Streaming) as well as the data quality checks with Great Expectations.

Where to Store Processed Streaming Data?

After Flink (or your chosen streaming solution) processes data from Kafka, you have two common options:

1. **Store in the Data Lake (MinIO/GCS):**
    - **Use Case:** This is useful if you want to retain the raw or lightly transformed streaming data for additional batch processing or historical archiving.
    - **Flow:** The streaming job writes its output to the data lake, and later a batch process can load that data into BigQuery staging tables.
2. **Store Directly in the Staging Data Warehouse (BigQuery):**
    - **Use Case:** If you aim for a unified pipeline where both batch and streaming data are immediately available for SQL-based transformations via dbt, then writing directly to staging tables in BigQuery is preferred.
    - **Flow:** The streaming job sends processed data straight into BigQuery staging tables. From there, dbt can run incremental models to update the final feature tables.

Recommendation:
For consistency and easier integration of batch and streaming pipelines, it's often best to load the processed streaming data directly into BigQuery staging tables. This approach

allows dbt to treat both data sources uniformly, apply incremental transformations, and run quality checks (with Great Expectations) on a single source of truth before the data is pushed to your feature store.

Summary of the Data Flow

1. **Data Ingestion:**
   - **Batch:** Local files are ingested by PySpark, processed, and written to the Data Lake.
   - **Streaming:** Debezium captures changes from the transactional DB and pushes events to Kafka. Flink consumes these events, applies real-time transformations, and writes the results either to the Data Lake or directly into BigQuery staging.
2. **Integration:**
   - Whether coming from batch or streaming, the processed data is loaded into BigQuery staging tables.
3. **Transformation:**
   - **dbt** then runs SQL-based transformations on these staging tables to build the final feature tables.
   - **Great Expectations** validates the data quality during or after these transformations.
4. **Orchestration:**
   - **Airflow** oversees the execution of all these jobs—scheduling PySpark batches, triggering streaming jobs, running dbt transformations, and executing data quality tests.
5. **Feature Serving:**
   - The final, curated feature tables are fed into Feast, which serves them for AI/ML training and real-time inference.

By centralizing the processed data in BigQuery staging, you achieve a unified, maintainable architecture where both batch and streaming pipelines contribute to the same high-quality feature store.

i want both store the raw streaming data on data lake and writes the consume results directly into BigQuery staging

You can design your streaming pipeline to have two parallel outputs:

- **Raw Data to Data Lake:**
  After consuming events from Kafka (via Flink, Spark Structured Streaming, or a similar engine), one branch writes the untouched, raw streaming data to your data lake (MinIO or GCS). This serves as a historical archive, enabling you to replay or reprocess the original events if needed.
- **Processed Data to BigQuery Staging:**
  Simultaneously, the streaming job applies the necessary transformations (using a shared transformation logic to ensure consistency with your batch pipeline) and writes the processed results directly into staging tables in BigQuery. From here, dbt can further transform and refine the data into final feature tables.

How It Works Together:

- **Ingestion & Processing:** Debezium captures CDC events from your transactional DB and publishes them to Kafka. A streaming engine (Flink/Spark Streaming) then reads from Kafka, splitting its processing into two sinks:
  - **Raw Sink:** Persists the original events to the data lake.
  - **Processed Sink:** Writes the cleaned/aggregated data to BigQuery staging.
- **Transformation with dbt:** Once the processed data is in BigQuery, dbt runs incremental models (triggered by Airflow) to create curated feature tables.
- **Orchestration & Quality Checks:** Airflow coordinates the streaming job, dbt runs, and any Great Expectations validations to ensure data quality throughout the process.

This dual-sink approach ensures that you both retain a complete, unaltered record of your streaming data and have immediate access to processed, transformation-ready data in BigQuery for building your AI/ML features.

Give me the final architecture, the final folder structure and describe each component and the data flow

Below is the final, comprehensive design that incorporates dual-sink streaming (raw data to the Data Lake and processed data directly to BigQuery staging), along with batch ingestion, transformations, orchestration, and quality checks. This answer includes an updated system architecture diagram, a final folder structure, and a detailed explanation of each component and data flow.

Final System Architecture Diagram

```plaintext
                               ┌─────────────────────────┐
                               │      Data Sources       │
                               │ ┌─────────────────────┐ │
                               │ │ Local Batch Files   │ │
                               │ └─────────────────────┘ │
                               │ ┌─────────────────────┐ │
                               │ │ Transactional DB    │ │
                               │ │ (for CDC events)    │ │
                               │ └─────────────────────┘ │
                               └────────────┬────────────┘
                                            │
                  ┌─────────────────────────┼─────────────────────────────┐
                  │                         │                             │
                  ▼                         ▼                             ▼
       ┌──────────────────────┐  ┌──────────────────────┐
   ┌──────────────────────┐
       │ PySpark Batch Jobs   │  │    Debezium          │        │   Airflow
   │
       │ (Ingestion &         │  │ (CDC from DB to      │        │ (Orchestration of
   │
       │  Transformations)    │  │  Kafka)              │        │  Batch & Streaming)
   │
       └───────────┬──────────┘  └───────────┬──────────┘
   └───────────┬──────────┘
                   │                          │                                  │
                   │                          ▼                                  │
                   │                ┌──────────────────────┐                       │
                   │                │        Kafka         │                       │
                   │                │ (Event Streaming)    │                       │
                   │                └───────────┬──────────┘                       │
                   │                            │                                  │
                   │                ┌───────────┴──────────────────┐               │
```

```
          │                   │  Streaming Engine          │                  │
          │                   │ (Flink / Spark Streaming)  │                  │
          │                   └────────────┬───────────────┘                  │
          │                                │                                  │
          │                ┌───────────────┴──────────────────┐               │
          │                │  Dual-Sink Output:                │               │
          │                │ ┌──────────────┐   ┌───────────┐  │  │              │
          │                │ │  Raw Data    │   │ Processed │  │  │              │
          │                │ │  to Data     │   │ Data to   │  │  │              │
          │                │ │  Lake        │   │ BigQuery  │  │  │              │
          │                │ │ (MinIO/GCS)  │   │ Staging   │  │  │              │
          │                │ └──────────────┘   └───────────┘  │  │              │
          │                └───────────────┬──────────────────┘               │
          │                                │                                  │
          │                                ▼                                  │
          │                     ┌──────────────────────┐                      │
          │                     │   Data Warehouse     │                      │
          │                     │   (BigQuery)         │                      │
          │                     │ (Staging Tables)     │                      │
          │                     └─────────┬────────────┘                      │
          │                               │                                   │
          │                               ▼                                   │
          │                     ┌──────────────────────┐                      │
          │                     │         dbt          │                      │
          │                     │ (SQL Transformations │                      │
          │                     │  & Feature Modeling) │                      │
          │                     └─────────┬────────────┘                      │
          │                               │                                   │
          │                               ▼                                   │
          │                     ┌──────────────────────┐                      │
          │                     │   Feature Store      │                      │
          │                     │      (Feast)         │                      │
          │                     └─────────┬────────────┘                      │
          │                               │                                   │
          │                               ▼                                   │
          └─────────────────────►┌───────────────────────────┐◄───────────────┘
                                 │  ML Models / Downstream   │
                                 │  Applications (Training & │
                                 │   Real-Time Inference)    │
                                 └───────────────────────────┘
```

Final Folder Structure

```bash
project-root/
├── README.md
├── docker/
│   ├── Dockerfile.airflow
│   ├── Dockerfile.flink
│   ├── Dockerfile.kafka
│   ├── Dockerfile.pyspark
│   ├── Dockerfile.feast
│   └── ...
├── infra/
│   ├── k8s/
│   │   ├── deployments/
│   │   │   ├── airflow-deployment.yaml
│   │   │   ├── flink-deployment.yaml
│   │   │   ├── kafka-deployment.yaml
│   │   │   ├── feast-deployment.yaml
│   │   │   └── ...
│   │   ├── services/
│   │   │   ├── airflow-service.yaml
│   │   │   ├── kafka-service.yaml
│   │   │   └── ...
│   │   └── configs/
│   │       ├── minio-configmap.yaml
│   │       └── ...
│   ├── terraform/
│   │   ├── main.tf
│   │   ├── variables.tf
│   │   └── ...
│   └── ...
├── configs/
│   ├── airflow/
│   │   └── airflow.cfg
│   ├── spark/
│   │   └── spark-defaults.conf
│   ├── flink/
│   │   └── flink-conf.yaml
│   ├── feast/
│   │   └── feature_store.yaml
```

```
│   ├── kafka/
│   │   └── server.properties
│   └── ...
├── scripts/
│   ├── setup_db.sh
│   ├── init_minio.sh
│   ├── mock_event_generation.py
│   └── ...
├── src/
│   ├── pyspark_jobs/
│   │   ├── batch_ingestion/
│   │   │   ├── main_batch.py
│   │   │   ├── transformations.py
│   │   │   └── ...
│   │   ├── streaming_ingestion/
│   │   │   ├── main_stream.py
│   │   │   ├── flink_jobs/
│   │   │   │   ├── kafka_consumer.py
│   │   │   │   └── ...
│   │   │   ├── pyspark_streaming/
│   │   │   │   └── ...
│   │   │   ├── cdc/
│   │   │   │   ├── debezium_config.json
│   │   │   │   └── ...
│   │   │   └── dual_sink_logic.py  # Implements writing raw & processed data
│   │   ├── feature_engineering/
│   │   │   ├── feature_calc.py
│   │   │   └── ...
│   │   └── utils/
│   │       ├── common_utils.py
│   │       └── ...
│   ├── data_validation/
│   │   └── great_expectations/
│   │       ├── expectations/
│   │       ├── checkpoints/
│   │       └── great_expectations.yml
│   └── feature_store/
│       ├── feast/
│       │   ├── feature_defs.py
│       │   └── registry.db
│       └── ...
├── orchestration/
```

```
│   └── airflow/
│       ├── dags/
│       │   ├── batch_dag.py
│       │   ├── streaming_dag.py
│       │   └── orchestration_dag.py  # Triggers streaming/batch jobs, dbt, and GE
tasks
│       └── plugins/
│           └── ...
├── dbt/
│   ├── dbt_project.yml
│   ├── models/
│   │   ├── staging/
│   │   │   ├── batch_staging.sql
│   │   │   ├── streaming_staging.sql
│   │   │   └── unified_staging.sql
│   │   ├── marts/
│   │   │   ├── feature_table.sql
│   │   │   └── ...
│   │   └── ...
│   └── seeds/
├── tests/
│   ├── unit/
│   │   ├── test_batch_ingestion.py
│   │   ├── test_streaming.py
│   │   └── ...
│   ├── integration/
│   │   ├── test_end_to_end.py
│   │   └── ...
│   └── ...
└── .github/workflows/
    ├── ci_cd_pipeline.yml
    └── ...
```

Component Descriptions & Data Flow

1. Data Sources

- **Local Batch Files:**
  Historical or periodic files (CSV, Parquet, etc.) are placed in a designated directory or

object storage. These files feed the batch processing pipeline.

- **Transactional Database:**
  This database captures real-time events (inserts, updates, deletes). Debezium monitors its change log to capture CDC events.

---

2. Ingestion Pipelines

- **Batch Processing (PySpark):**
  - **Role:** Reads raw local batch files, applies complex distributed transformations and feature engineering.
  - **Output:** Writes processed data to the Data Lake (MinIO or GCS) and/or loads it into BigQuery staging tables.
- **Streaming Processing (Debezium → Kafka → Flink/Spark Streaming):**
  - **Debezium:** Monitors the transactional DB and publishes CDC events to Kafka.
  - **Kafka:** Serves as the central event streaming platform.
  - **Streaming Engine (Flink/Spark Streaming):**
    - **Dual-Sink Output:**
      1. **Raw Data Sink:** Writes the unaltered streaming events to the Data Lake for archival and potential reprocessing.
      2. **Processed Data Sink:** Applies incremental transformations (using shared logic with batch jobs) and writes the clean, processed data directly into BigQuery staging tables.

---

3. Data Storage Layers

- **Data Lake (MinIO/GCS):**
  - **Purpose:** Acts as a landing zone for all raw data—both batch and streaming.
  - **Usage:** Stores raw streaming events, allowing reprocessing if needed.

- **Data Warehouse (BigQuery):**
  - **Purpose:** Centralized staging area where both processed batch and streaming data are loaded.
  - **Usage:** Serves as the input for further SQL-based transformations via dbt.

---

4. Transformations with dbt

- **dbt:**
  - **Role:** Executes SQL transformations in BigQuery to turn staging tables into clean, curated feature tables.
  - **Key Features:**
    - **Incremental Models:** Processes only new or updated data from both batch and streaming sources.
    - **Data Modeling:** Creates well-structured marts and dimensions that feed the feature store.
  - **Outcome:** The final feature tables are analytics-ready and accessible for ML training and inference.

---

5. Feature Serving (Feast)

- **Feast:**
  - **Role:** Centralized feature store that unifies offline (training) and online (real-time) features.
  - **Data Sources:**
    - **Offline Store:** Pulls refined features from BigQuery.
    - **Online Store:** Can be updated from BigQuery or via streaming outputs for real-time serving.

---

6. Orchestration & Quality

- **Airflow:**
  - **Role:** Orchestrates the entire pipeline—scheduling PySpark batch jobs, triggering streaming pipelines, running dbt transformations, and coordinating Great Expectations data quality checks.
  - **DAGs:** Define separate workflows for batch ingestion, streaming ingestion, and transformation tasks.
- **Great Expectations:**
  - **Role:** Validates data quality at critical checkpoints (e.g., post-ingestion, pre- and post-dbt transformations).
  - **Integration:** Runs as part of Airflow DAGs or standalone jobs to ensure the integrity of both raw and processed data.

---

7. Downstream Consumption

- **ML Models & Applications:**
  - **Usage:** Access the final feature tables from BigQuery or retrieve features in real time from Feast's online store for model training and inference.

Overall Data Flow Summary

1. **Data Ingestion:**
   - Batch files are processed by PySpark; CDC events from the transactional DB are captured by Debezium and sent to Kafka.
2. **Streaming Processing:**
   - A streaming engine (Flink/Spark Streaming) consumes Kafka events and writes:

- **Raw events** to the Data Lake.
- **Processed events** directly to BigQuery staging.

3. **Transformation:**
   - dbt reads staging tables in BigQuery (populated by both batch and streaming) to build refined, incremental feature tables.
4. **Feature Serving:**
   - The final features are stored in Feast for both offline analysis and real-time scoring.
5. **Orchestration & Quality Control:**
   - Airflow coordinates all jobs, and Great Expectations ensures data quality throughout the pipeline.
6. **Consumption:**
   - ML models and downstream applications use these features for training and inference.

This final architecture, with its clear folder structure and detailed component descriptions, ensures a scalable, maintainable pipeline that handles both raw and processed streaming data alongside batch data—delivering consistent, high-quality features for your AI/ML applications.

what is the purpose of folder "feature_engineering", I thought DBT already did that to produce features tables. Where are complex feature transformations performed?

The **dbt** layer is primarily used for SQL-based transformations—organizing, cleaning, and modeling data that has already landed in BigQuery. In contrast, the **feature_engineering** folder is intended for more complex or custom transformations that might not be easily expressed in SQL.

When to Use the Feature Engineering Folder:

- **Advanced Computations:**
  If you need to perform transformations that require custom Python logic, advanced libraries, or distributed computing (e.g., using PySpark's ML or window functions that are better implemented in code), those tasks can be handled in this folder.
- **Complex Feature Extraction:**
  For feature transformations that involve non-trivial algorithms, heavy data processing,

or iterative computations, it’s often more efficient to do this in a code environment (such as PySpark) rather than in SQL.

- **Pre-Processing Before dbt:**
  The outputs from the feature_engineering code can be written to staging areas (like BigQuery staging tables) where **dbt** can then perform further refinements, testing, and modeling.

Summary:

- **dbt:** Transforms and models data in BigQuery using SQL, ideal for standard incremental updates, cleaning, and creating final feature tables.
- **Feature Engineering Folder:** Contains code for performing advanced, complex transformations that require a more flexible programming environment than SQL. This might be part of your PySpark pipeline or another dedicated processing step.

In this way, the **feature_engineering** folder complements dbt by handling tasks that go beyond standard SQL transformations, ensuring that all necessary feature calculations—from the most basic to the most complex—are efficiently processed before the final data modeling step.

should i use Flink or Spark Streaming ?

Both Apache Flink and Spark Structured Streaming have their merits, and the right choice depends on your specific requirements. Here’s a quick comparison to help you decide:

When to Consider Spark Structured Streaming

- **Unified API:**
  If you already use PySpark for batch processing, leveraging Spark Structured Streaming lets you reuse the same APIs and infrastructure for both batch and streaming workloads.
- **Micro-Batching Model:**
  Spark Structured Streaming processes data in small micro-batches, which can be sufficient if your use case can tolerate a slight delay in processing and if your transformations are largely similar to batch jobs.
- **Ecosystem Integration:**
  When your pipeline is heavily integrated with Spark, using Structured Streaming can simplify deployment and operational overhead.

When to Consider Apache Flink

- **True Streaming Semantics:**
  Flink offers native, event-by-event processing with lower latencies and true streaming semantics. This is especially beneficial if you require fine-grained control over event time, windowing, or exactly-once processing guarantees.
- **Complex Event Processing:**
  Flink shines in scenarios where you need complex event processing or stateful streaming with minimal latency.
- **Flexibility:**
  If your streaming logic involves highly dynamic or complex transformations that require low latency processing, Flink may offer more flexibility and performance.

Recommendation

- **If you have a unified Spark ecosystem and can work with micro-batching:**
  Use **Spark Structured Streaming** for a seamless integration with your existing PySpark batch jobs.
- **If low latency and true event-by-event processing are critical to your application:**
  Choose **Apache Flink** for its native streaming capabilities and advanced event processing features.

Ultimately, the choice depends on your latency requirements, existing infrastructure, and the complexity of your streaming transformations. Both are robust, so evaluating these factors in the context of your project's needs will lead you to the best option.

should i have a .env file ? if yes, should its place ? how can i design for difference seperate enviroment like dev and prod

Yes, it's a good practice to use a **.env** file to manage environment-specific configuration values (e.g., database URLs, API keys, secrets). This keeps sensitive data and configuration out of your source code and makes it easier to manage different settings for development, testing, and production.

Where to Place the .env File?

- **Project Root:**
  The .env file is typically placed in the root directory of your project. This allows your application code to load environment variables consistently, regardless of the subdirectory structure.
- **Security:**
  Make sure to add the .env file to your .gitignore so that it doesn't get committed to version control, protecting sensitive information.

Designing for Separate Environments (Dev, Prod, etc.)

1. **Multiple .env Files:**
   - You can create different files for each environment, such as:
     - `.env.dev` for development.
     - `.env.test` for testing.
     - `.env.prod` for production.
   - Then, during deployment or startup, load the correct file based on an environment variable (e.g., `ENV=dev` or `ENV=prod`).
2. **Environment Variable Management:**
   - Use libraries like `python-dotenv` (for Python projects) or similar tools in other languages to load environment variables at runtime.
   - Your deployment process (e.g., CI/CD pipelines, Kubernetes ConfigMaps/Secrets) can also inject environment variables directly, reducing reliance on static files.
3. **Configuration Abstraction:**
   - Consider creating a configuration module or class in your application that reads the environment variables and provides a structured configuration object. This centralizes configuration logic and helps manage differences between environments.

Example Setup

```bash
project-root/
├── .env.dev         # Development environment variables
├── .env.prod        # Production environment variables
```

```
├── .gitignore        # Make sure to ignore .env* files
├── README.md
├── src/
│   ├── app.py        # Application code that loads environment variables
│   └── config.py     # Configuration module that reads the .env file based on the
current ENV
└── ...
```

In `config.py`:

```python
import os
from dotenv import load_dotenv

# Determine which .env file to load based on an environment variable
env = os.getenv('ENV', 'dev')
dotenv_path = f'.env.{env}'
load_dotenv(dotenv_path)

# Now access environment variables
DATABASE_URL = os.getenv('DATABASE_URL')
API_KEY = os.getenv('API_KEY')
# ... other configuration variables
```

In Your Deployment/CI/CD:

- Set the `ENV` variable to `prod` (or another appropriate value) so that the correct .env file is loaded.
- Alternatively, for production, you might inject variables directly via your orchestrator (like Kubernetes Secrets) rather than using a .env file.

Summary

- **Purpose:** .env files store configuration and secrets separately from your code.
- **Placement:** Place them at the project root and exclude from version control.
- **Multi-Environment Design:** Use separate .env files (e.g., .env.dev, .env.prod) and a configuration loader that selects the right file based on an environment variable. This approach simplifies development and ensures that your application runs with the correct settings in different environments.

This strategy allows your application to adapt to various environments with minimal changes to your codebase and supports secure, maintainable configuration management.

create me a drawio architecture

Below is an example of a Draw.io (diagrams.net) XML file that you can import to visualize the final architecture diagram. This diagram includes data sources, batch and streaming ingestion (with dual-sink output), orchestration, transformations (dbt), feature serving (Feast), and downstream ML consumption. You can customize the positions, sizes, or colors as needed.

---

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mxfile host="app.diagrams.net" modified="2025-02-14T10:00:00.000Z" agent="ChatGPT"
version="14.6.13">
  <diagram id="finalArchitecture" name="Final Architecture">
    <mxGraphModel dx="1074" dy="662" grid="1" gridSize="10" guides="1" tooltips="1"
connect="1" arrows="1" fold="1" page="1" pageScale="1" pageWidth="1169"
pageHeight="827">
      <root>
        <mxCell id="0"/>
        <mxCell id="1" parent="0"/>
        <!-- Data Sources -->
        <mxCell id="2" value="Data Sources"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#DAE8FC;strokeColor=#6C8EBF;fontSize=
vertex="1" parent="1">
          <mxGeometry x="20" y="20" width="140" height="80" as="geometry"/>
        </mxCell>
        <mxCell id="3" value="Local Batch Files"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFFFFF;strokeColor=#000000;fontSize=
vertex="1" parent="2">
          <mxGeometry x="10" y="40" width="120" height="30" as="geometry"/>
        </mxCell>
        <mxCell id="4" value="Transactional DB (CDC)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFFFFF;strokeColor=#000000;fontSize=
vertex="1" parent="2">
          <mxGeometry x="10" y="75" width="120" height="30" as="geometry"/>
```

```xml
        </mxCell>
        <!-- Batch Ingestion (PySpark) -->
        <mxCell id="5" value="PySpark Batch Jobs"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFF2CC;strokeColor=#D6B656;fontSize=
vertex="1" parent="1">
          <mxGeometry x="200" y="20" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- Debezium -->
        <mxCell id="6" value="Debezium"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFF2CC;strokeColor=#D6B656;fontSize=
vertex="1" parent="1">
          <mxGeometry x="200" y="100" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- Airflow (Orchestration) -->
        <mxCell id="16" value="Airflow (Orchestration)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFE6CC;strokeColor=#D79B00;fontSize=
vertex="1" parent="1">
          <mxGeometry x="200" y="200" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- Kafka -->
        <mxCell id="7" value="Kafka"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#E1D5E7;strokeColor=#9673A6;fontSize=
vertex="1" parent="1">
          <mxGeometry x="400" y="100" width="140" height="60" as="geometry"/>
        </mxCell>
        <!-- Streaming Engine (Flink/Spark Streaming) -->
        <mxCell id="8" value="Streaming Engine (Flink/Spark Streaming)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#E1D5E7;strokeColor=#9673A6;fontSize=
vertex="1" parent="1">
          <mxGeometry x="600" y="100" width="220" height="60" as="geometry"/>
        </mxCell>
        <!-- Dual-Sink Output -->
        <mxCell id="9" value="Dual-Sink Output"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#F8CECC;strokeColor=#B85450;fontSize=
vertex="1" parent="1">
          <mxGeometry x="850" y="100" width="160" height="80" as="geometry"/>
        </mxCell>
        <mxCell id="10" value="Raw Data to Data Lake (MinIO/GCS)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFFFFF;strokeColor=#000000;fontSize=
vertex="1" parent="9">
          <mxGeometry x="10" y="20" width="140" height="30" as="geometry"/>
        </mxCell>
```

```
        <mxCell id="11" value="Processed Data to BigQuery Staging"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#FFFFFF;strokeColor=#000000;fontSize=
vertex="1" parent="9">
          <mxGeometry x="10" y="50" width="140" height="30" as="geometry"/>
        </mxCell>
        <!-- BigQuery (Data Warehouse) -->
        <mxCell id="12" value="Data Warehouse (BigQuery)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#D5E8D4;strokeColor=#82B366;fontSize=
vertex="1" parent="1">
          <mxGeometry x="1050" y="100" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- dbt -->
        <mxCell id="13" value="dbt (SQL Transformations)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#D5E8D4;strokeColor=#82B366;fontSize=
vertex="1" parent="1">
          <mxGeometry x="1050" y="200" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- Feature Store (Feast) -->
        <mxCell id="14" value="Feature Store (Feast)"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#D5E8D4;strokeColor=#82B366;fontSize=
vertex="1" parent="1">
          <mxGeometry x="1050" y="300" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- ML Models / Downstream Apps -->
        <mxCell id="15" value="ML Models / Downstream Apps"
style="rounded=1;whiteSpace=wrap;html=1;fillColor=#DDEBF7;strokeColor=#6C8EBF;fontSize=
vertex="1" parent="1">
          <mxGeometry x="1050" y="400" width="160" height="60" as="geometry"/>
        </mxCell>
        <!-- Edges -->
        <mxCell id="edge1"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="2" target="5">
          <mxGeometry relative="1" as="geometry"/>
        </mxCell>
        <mxCell id="edge2"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="2" target="6">
          <mxGeometry relative="1" as="geometry"/>
        </mxCell>
        <mxCell id="edge3"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
```

```
parent="1" source="6" target="7">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge4"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="7" target="8">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge5"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="8" target="9">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge6"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="9" target="12">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge7"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="12" target="13">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge8"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="13" target="14">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge9"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="14" target="15">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge10"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="5" target="12">
            <mxGeometry relative="1" as="geometry"/>
          </mxCell>
          <mxCell id="edge11"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="16" target="5">
            <mxGeometry relative="1" as="geometry"/>
```

```
        </mxCell>
        <mxCell id="edge12"
style="edgeStyle=orthogonalEdgeStyle;rounded=0;html=1;strokeColor=#000000;" edge="1"
parent="1" source="16" target="6">
          <mxGeometry relative="1" as="geometry"/>
        </mxCell>
      </root>
    </mxGraphModel>
  </diagram>
</mxfile>
```

How to Use This XML

1. **Copy the XML code above.**
2. **Go to** draw.io (or open your Draw.io desktop application).
3. **Select "File" → "Import From" → "Device"** and paste the XML code (or save it as an .xml file and import it).
4. **The diagram will render with all components and connections.**

Diagram Components Recap

- **Data Sources:**
  - *Local Batch Files* and *Transactional DB (CDC)* are the starting points.
- **Ingestion Pipelines:**
  - *PySpark Batch Jobs* process batch data.
  - *Debezium* captures CDC events, sending them to *Kafka*.
- **Streaming Engine:**
  - *Streaming Engine (Flink/Spark Streaming)* consumes Kafka events.
- **Dual-Sink Output:**
  - One branch writes raw data to the *Data Lake (MinIO/GCS)*,

    - while the other writes processed data to *BigQuery Staging*.
- **Data Warehouse & Transformation:**
    - Processed data in BigQuery is further transformed by *dbt*.
- **Feature Serving:**
    - Curated features are stored in *Feature Store (Feast)*.
- **Consumption:**
    - *ML Models / Downstream Apps* use the features.
- **Orchestration:**
    - *Airflow* orchestrates the batch and streaming pipelines.

This Draw.io architecture diagram provides a visual overview of the entire data pipeline from ingestion to feature serving, making it easier to understand and communicate your system's design.