🧪 ETL Fundamentals for AI Pipelines

Generative AI Bootcamp – Week 2, Day 3, Session 1

November 26, 2025

🎯 Learning Objectives

- Understand ETL/ELT patterns across **batch** and **streaming**
- Map raw → clean → ready data for prompts and retrieval
- Choose the **right engine** for the **right scale/use-case**
- Identify quality checks and lineage practices
- See a concrete API → table → vector store pipeline

🧭 ETL vs ELT

- **ETL:** Extract → Transform → Load (classic DW)
- **ELT:** Extract → Load → Transform (cloud-first, push-down to warehouses/lakes)
- AI context: prioritize **fast ingestion** + **iterative transforms** for prompting and eval

🧱 Pipeline Building Blocks

- **Extract:** APIs, files, event streams (Kafka, Kinesis, Pub/Sub)
- **Transform:** cleaning, normalization, token-aware trimming, embedding
- **Load:** SQL/NoSQL/vector stores; partitioning & metadata

🧩 Data for LLMs

- Structured: metadata, eval metrics, user ids
- Semi-structured: JSON prompts, configs
- Unstructured: docs, transcripts, embeddings
- Keep **provenance** and **versioning** for reproducibility

🧼 Common Transformations

- Type casting & normalization
- Deduplication (by key / text hash)
- Text cleaning (HTML, emojis, whitespace)
- PII scrubbing & anonymization
- Token-length gating for context windows

🛡️ Data Quality & Validation

- Required columns present
- Non-null checks & allowed ranges
- Referential integrity across tables
- Assertions locally; **scalable checks** in orchestrated runs
- Tools: **Great Expectations, Deequ, Pandera, Pydantic**

🧰 Transform Engines — pick by scale & latency

Key heuristic: **push compute to where the data lives** (warehouse or lake).

1. Local / Small

For development or when <~1–5M rows.

- **pandas**: single machine, memory-bound
- **DuckDB**: SQL-in-process, great for Parquet/CSV joins
- **Polars**: fast, multi-threaded; lazy queries

🧰 Transform Engines — pick by scale & latency

2. Medium

Single host or small cluster.

- **Dask / Ray Data** (scale pandas-like APIs across cores/nodes)
- **Spark on a small cluster** for wide joins / shuffles

🧰 Transform Engines — pick by scale & latency

3. Large

Cluster, lake/warehouse.

- **Apache Spark / Databricks**
- **BigQuery / Snowflake / Redshift (ELT w/ SQL)**
- **Flink / Beam** for stateful & streaming with batch unification

🛰️ Streaming vs Batch

- **Batch**: scheduled, backfills, reprocessing large historical sets
- **Streaming**: low-latency updates, CDC, online features
- Buses: **Kafka, Kinesis, Pub/Sub**; Engines: **Flink, Beam, Spark Structured Streaming, Kafka Streams**

Examples:

- Near-real-time moderation → **Flink/Beam**
- Hourly RAG index refresh → **Spark or warehouse SQL**
- Heavy backfills → **Spark/Databricks** or **BigQuery/Snowflake** SQL

🗃️ Storage Targets

- **OLAP / Warehouses**: BigQuery, Snowflake, Redshift
- **Data Lakes / Lakehouses**: S3, GCS + Delta, Apache Iceberg, Hudi
- **OLTP / Application DBs**: Postgres, MySQL, MongoDB
- **Search / Vector**: Elasticsearch/OpenSearch, **pgvector**, FAISS, Milvus, Pinecone, Weaviate

🧭 Transform Authoring

- **SQL-first ELT** (dbt, SQLMesh, Dataform) → warehouse-native, declarative, tested
- **Python-first** (pandas, Polars, Dask, Spark) → code-based, complex or text-heavy logic
- **Unified batch/stream** (Beam, Flink) → for low-latency or event-based ETL

🧬 Lineage & Metadata

- Track: source URL, fetch time, schema version, transformation hash
- Emit OpenLineage events; catalog with **DataHub**, **Marquez**, **Amundsen**
- Store columns like `_ingested_at`, `_source`, `_version`

🎛️ Orchestration & Scheduling

- **Airflow**: mature DAGs, strong ecosystem
- **Prefect**: Pythonic flows, great observability
- **Dagster**: software-defined assets, type-checked IO
- Add **retries**, **backoff**, **idempotent** loads (UPSERT/MERGE)

📏 When to Use What (quick guide)

- **Notebook exploration / small batch** → pandas / Polars / DuckDB
- **Weekly/Monthly heavy batch** → Spark on lakehouse; or ELT in BigQuery/ Snowflake (dbt)
- **Hourly incremental** → warehouse MERGE with partitions or Spark jobs
- **Streaming / <1 min latency** → Flink/Beam/Kafka Streams
- **Text-heavy LLM prep** → Spark + UDFs, or Polars for speed; validate with Great Expectations

⚙️ Demo Pipeline (Today) — updated

- **Extract:** REST API → JSON (posts/comments) or Kafka topic
- **Transform (choose by size):**
  - Small: DuckDB/Polars → clean, normalize, validate (Pandera/GE)
  - Medium: Dask/Ray → parallel cleaning + joins
  - Large: Spark SQL/DataFrames → tokenization & embeddings at scale
- **Load:**
  - Warehouse tables (BigQuery/Snowflake) for curated data
  - Vector index (pgvector/FAISS/Pinecone) for retrieval
- **Schedule:** Prefect or Airflow; tests via dbt or GE

📦 Staging → Warehouse

- Use staging tables for raw dumps
- Run deterministic transforms into curated schemas
- Idempotent loads (UPSERT/MERGE by primary key)

⚠️ Pitfalls (now with scale in mind)

- Silent schema drift → contracts/tests (dbt/GE), schema registry for streams
- Overusing `SELECT *` → breakage on drift & higher costs
- No UTF-8/text normalization → broken tokenization
- Underestimating **shuffle** costs in distributed engines
- Forgetting incremental strategies & partition pruning

💡 Summary

- Choose the **engine by data size and latency** needs
- Validate early; preserve provenance and lineage
- Prefer push-down ELT in warehouses when possible
- Keep pipelines **modular, idempotent, observable**

🏁 What’s Next

- Hands-on: Build the API/stream → curated table → vector store ETL with two variants:
  - Small data: DuckDB + dbt seeds + GE
  - Large data: Spark + dbt models + Prefect