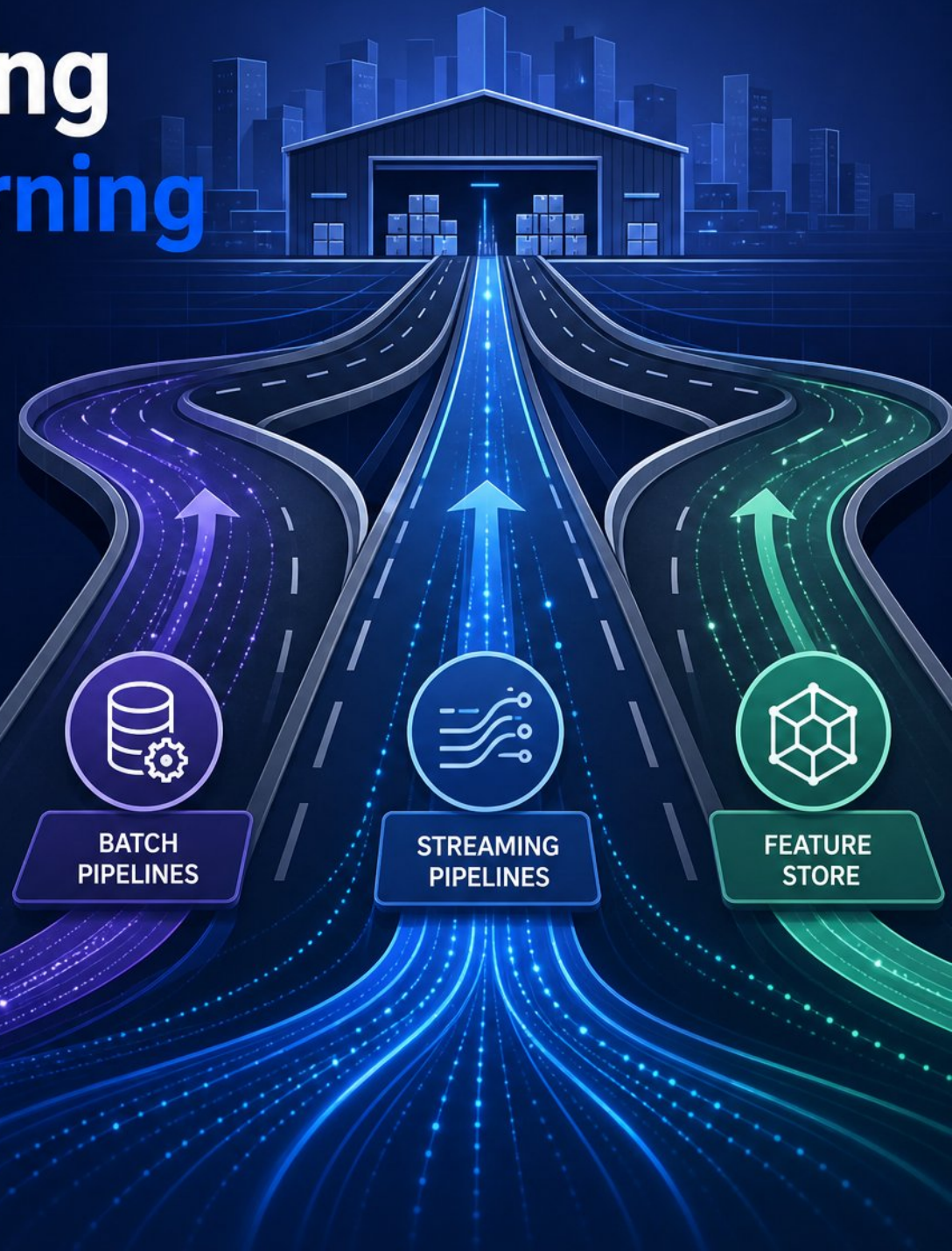
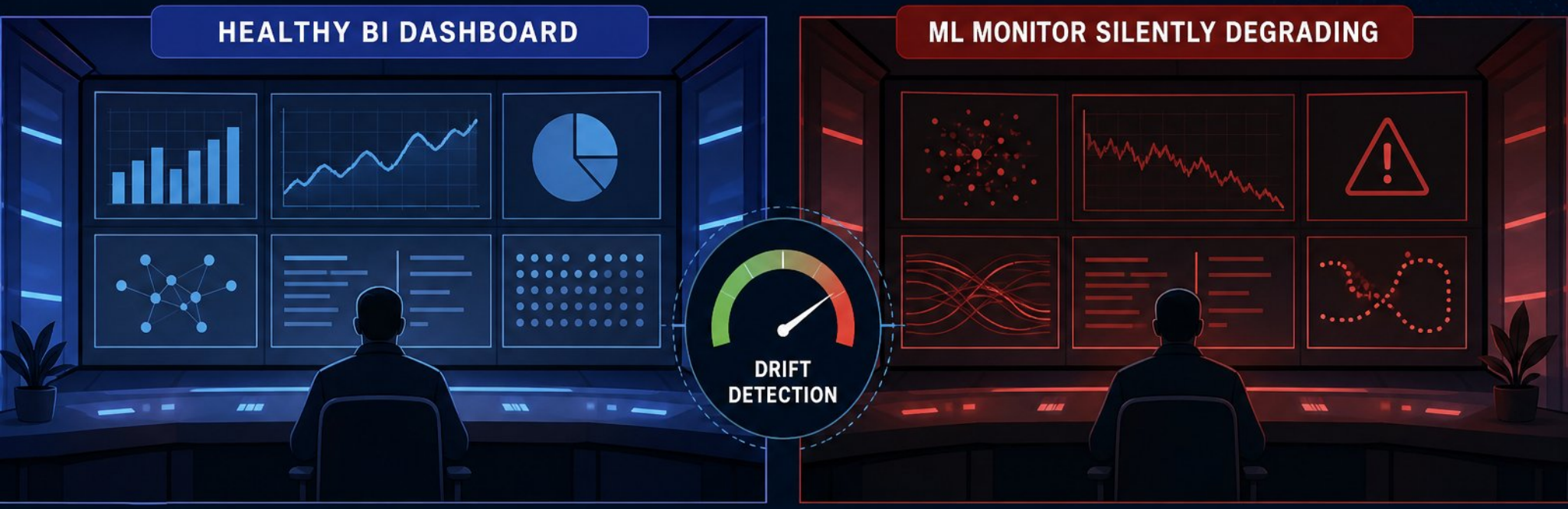
# Data Engineering
# for Machine Learning

- Extends traditional ETL beyond dashboards to model-ready data
- Spans ingestion, validation, transformation, features, serving, monitoring
- AI-ready data demands stricter quality, lineage, and freshness
- 60% of ML projects fail from data foundation gaps
- Course builds reliable, reproducible pipelines for data and ML teams

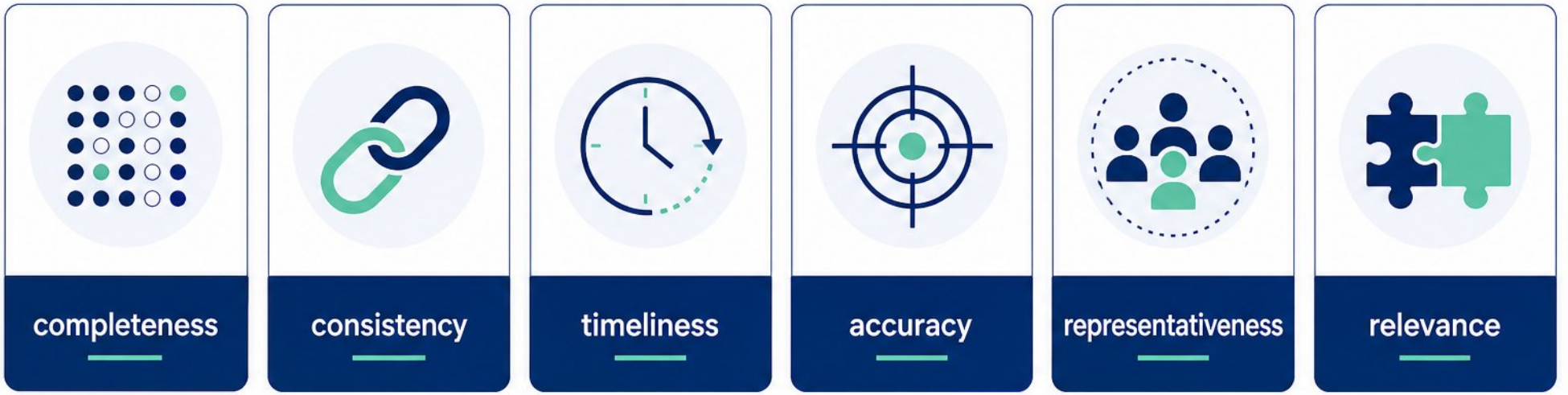


# Why ML Data Systems Fail Differently




- Wrong dashboard values are visible; model degradation is often silent
- Treating data engineering as an afterthought increases production ML failures
- Core anti-patterns: duplicated feature logic, uncontrolled schema changes, missing lineage
- Training goals: reliability, reproducibility, and point-in-time correctness

# ML Data Requirements and Quality Dimensions



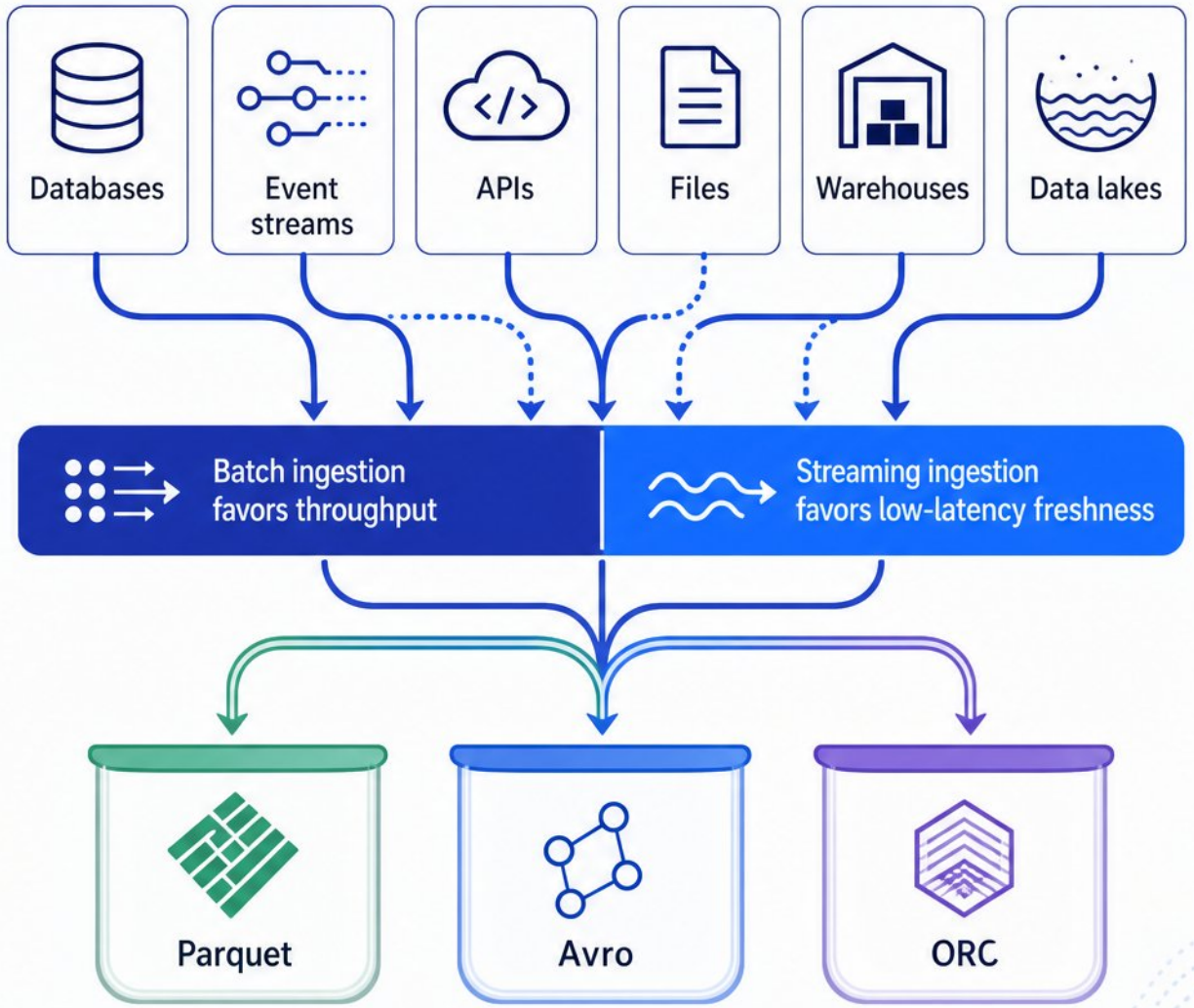

- Core dimensions: completeness, consistency, timeliness, accuracy, representativeness, relevance
- ML data quality adds a statistical layer—not just operational cleanliness
- Bias, leakage, and label quality are data engineering responsibilities
- Validate with schema, semantic, relational, and provenance checks

# Data Contracts and Pre-Training Validation

- Data contracts are versioned agreements between producers and ML consumers
- Contracts make silent upstream changes fail loudly and early
- Automate checks for schema, range, nulls, duplicates, and cross-field rules
- Validation gates block training jobs before bad data reaches the model
- CI/CD integration turns data quality into a pipeline guardrail

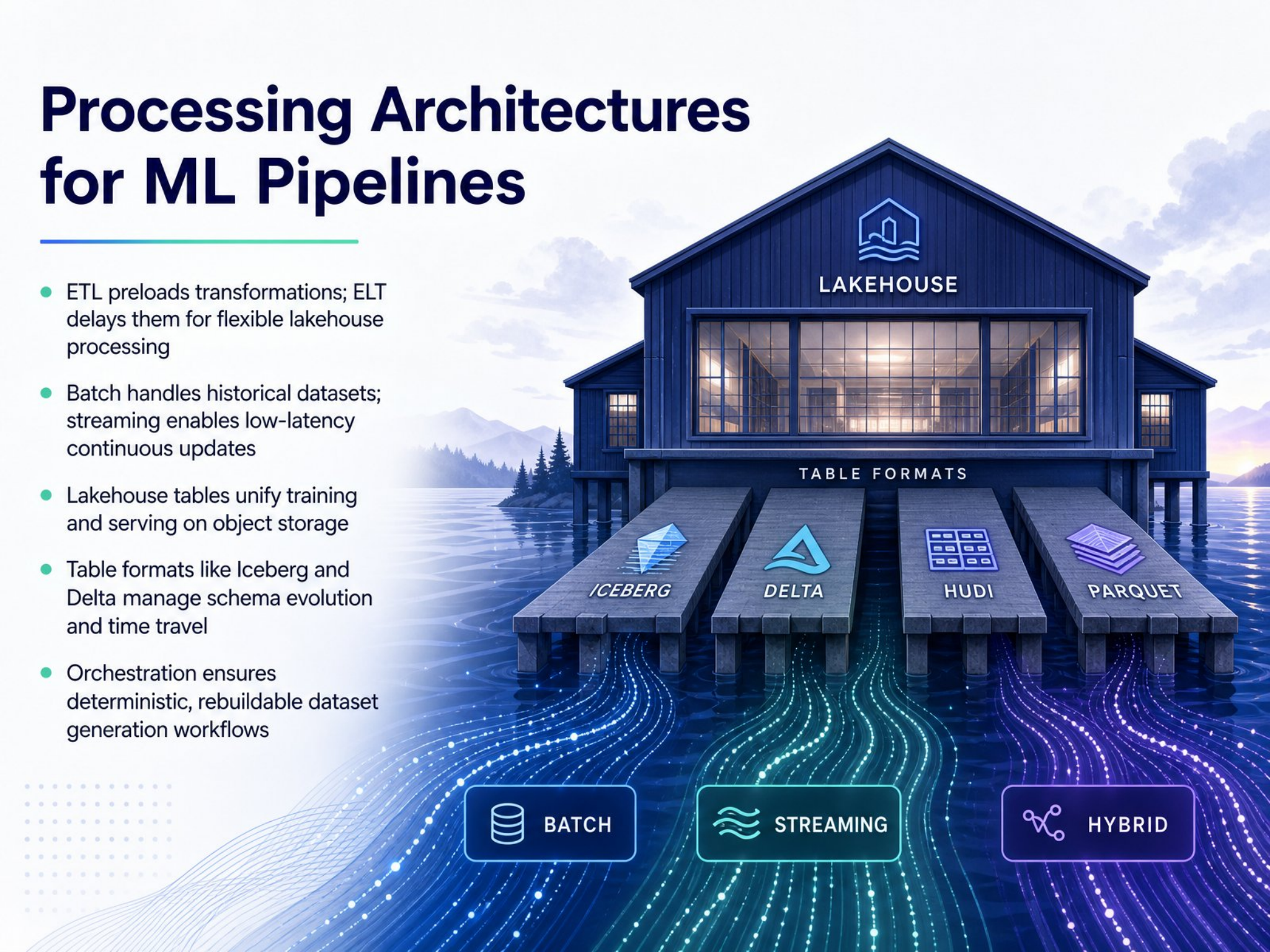
# Sources, Ingestion, and Storage Formats

- Databases, event streams, APIs, files, warehouses, and data lakes
- Batch ingestion favors throughput; streaming favors low-latency freshness
- Columnar formats (Parquet, ORC) optimize analytics and feature scans
- Row-based Avro suits streaming, writes, and schema evolution
- Use Parquet for training data, Avro for event pipelines



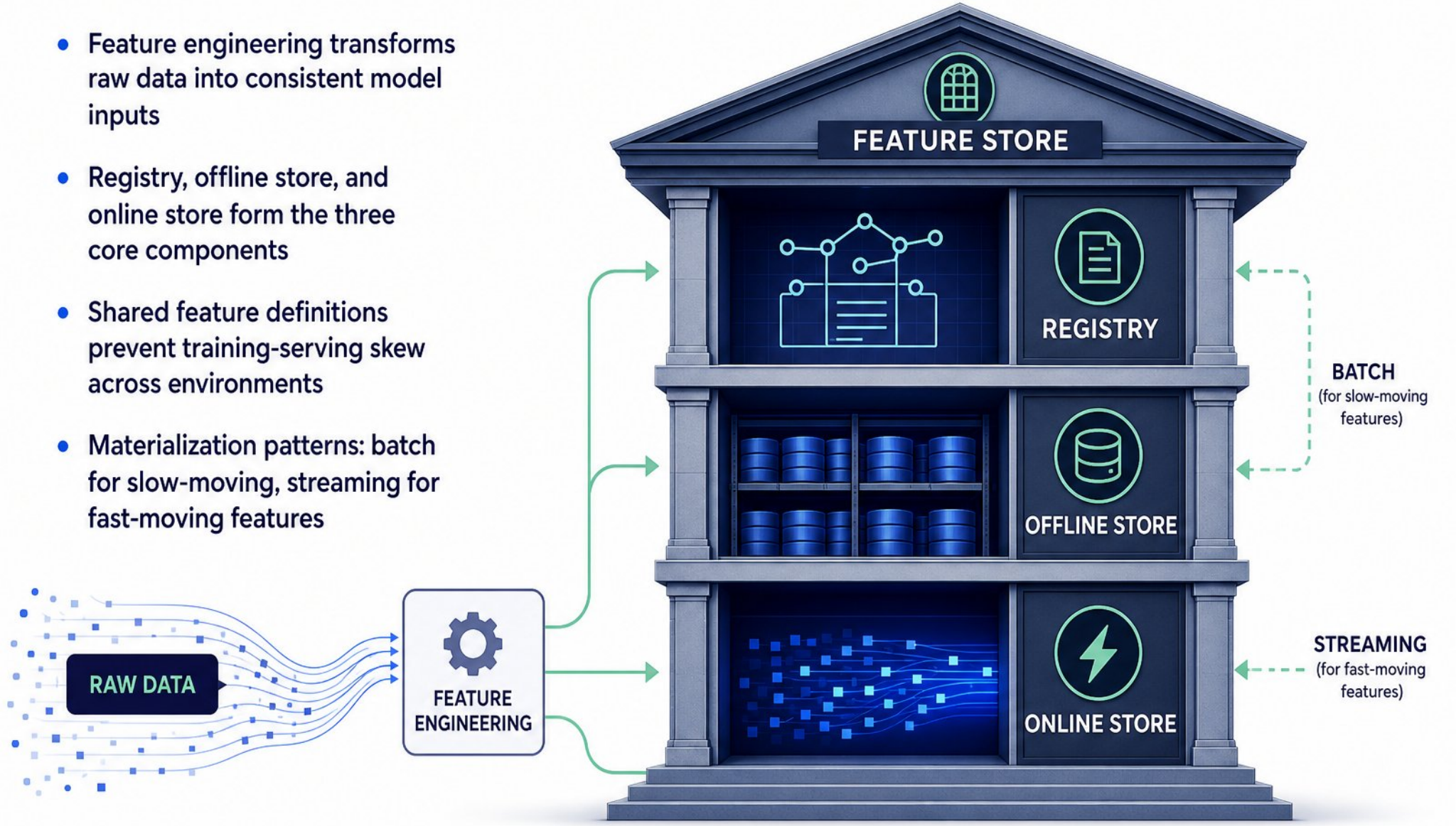
# Processing Architectures for ML Pipelines

- ETL preloads transformations; ELT delays them for flexible lakehouse processing
- Batch handles historical datasets; streaming enables low-latency continuous updates
- Lakehouse tables unify training and serving on object storage
- Table formats like Iceberg and Delta manage schema evolution and time travel
- Orchestration ensures deterministic, rebuildable dataset generation workflows



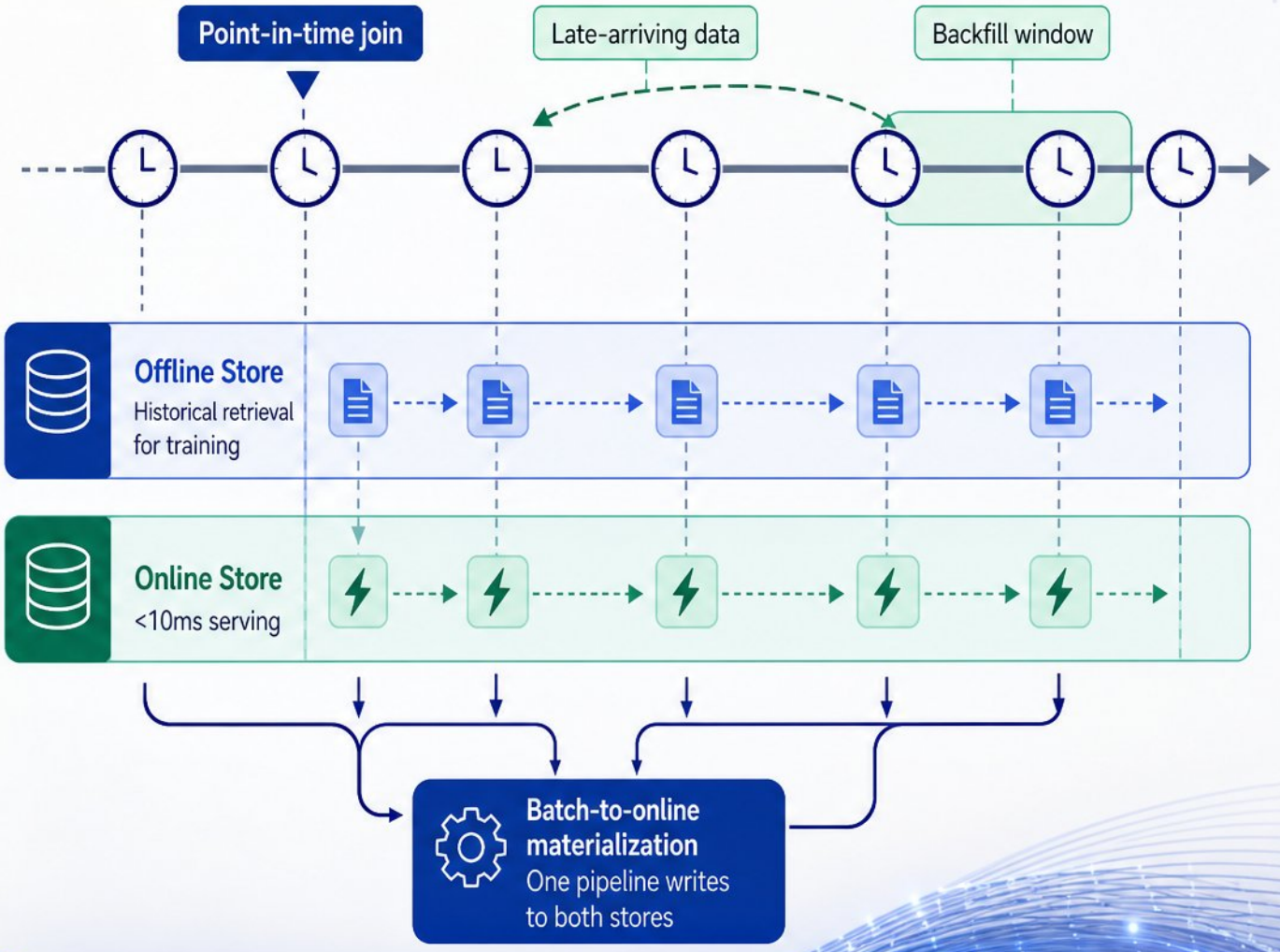
# Feature Engineering and Feature Stores

- Feature engineering transforms raw data into consistent model inputs
- Registry, offline store, and online store form the three core components
- Shared feature definitions prevent training-serving skew across environments
- Materialization patterns: batch for slow-moving, streaming for fast-moving features



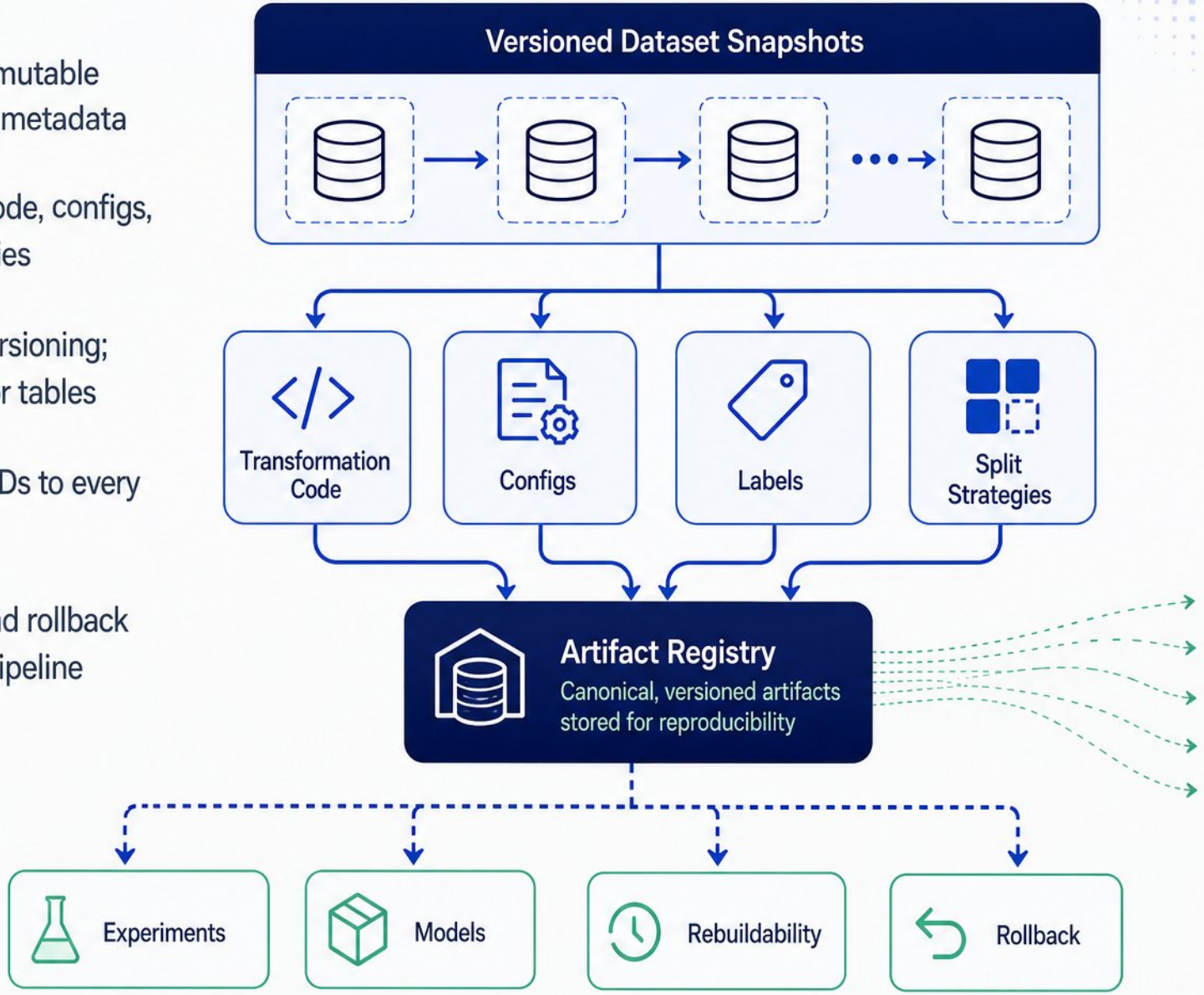
# Point-in-Time Correctness and Serving Consistency

- Point-in-time joins prevent leakage: features match each example's timestamp
- Offline store: historical retrieval for training; online store: <10ms serving
- Batch-to-online materialization: one pipeline writes to both stores
- Freshness SLAs per feature: batch for slow, streaming for fast updates
- Handle late-arriving data, backfills, and consistency tests between paths



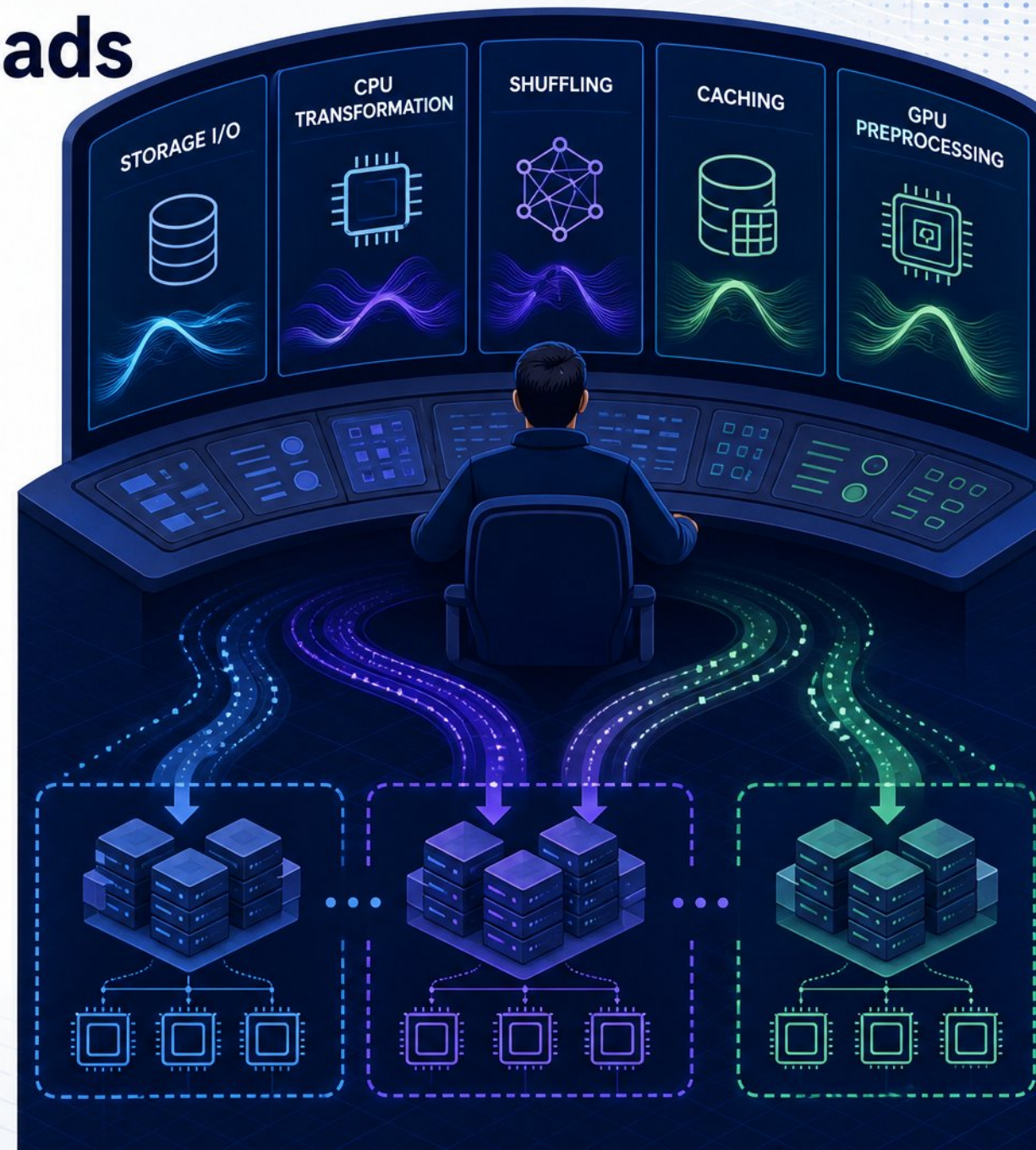
# Reproducible Training Data Pipelines

- Version datasets as immutable snapshots with lineage metadata
- Track transformation code, configs, labels, and split strategies
- Use DVC for Git-like versioning; lakehouse time travel for tables
- Attach dataset version IDs to every experiment and model
- Enable rebuildability and rollback through deterministic pipeline definitions



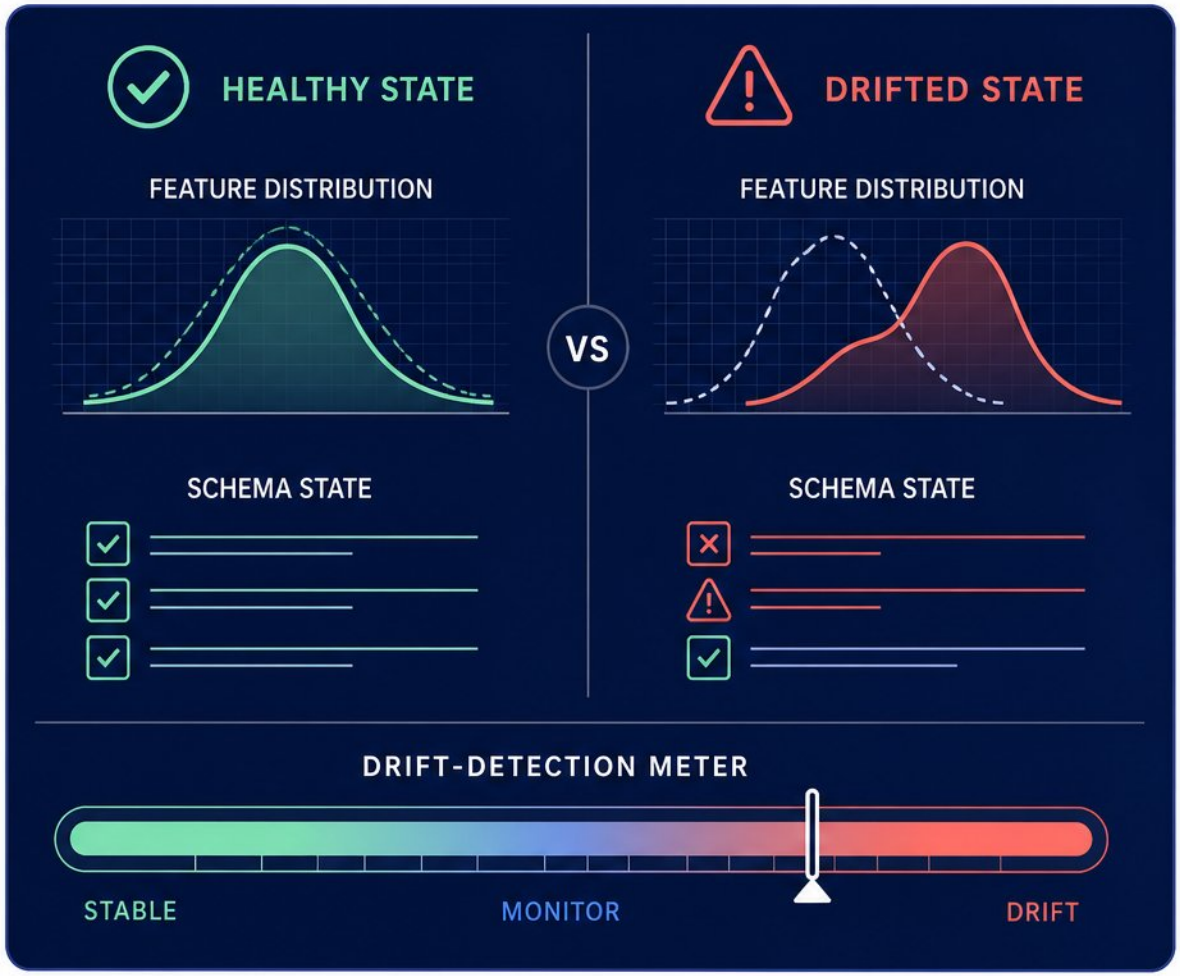
# Scaling Distributed ML Data Workloads

- Profile bottlenecks first: storage I/O, CPU transformation, and data loading
- Partition with macrosharding and microsharding across machines and vCPUs
- Compare distributed backends: Spark, Ray, GPU-native preprocessing, tf.data service
- Apply push-down transformations and local caching to reduce redundant I/O
- Handle uneven workloads with coordinated reads and dynamic autoscaling



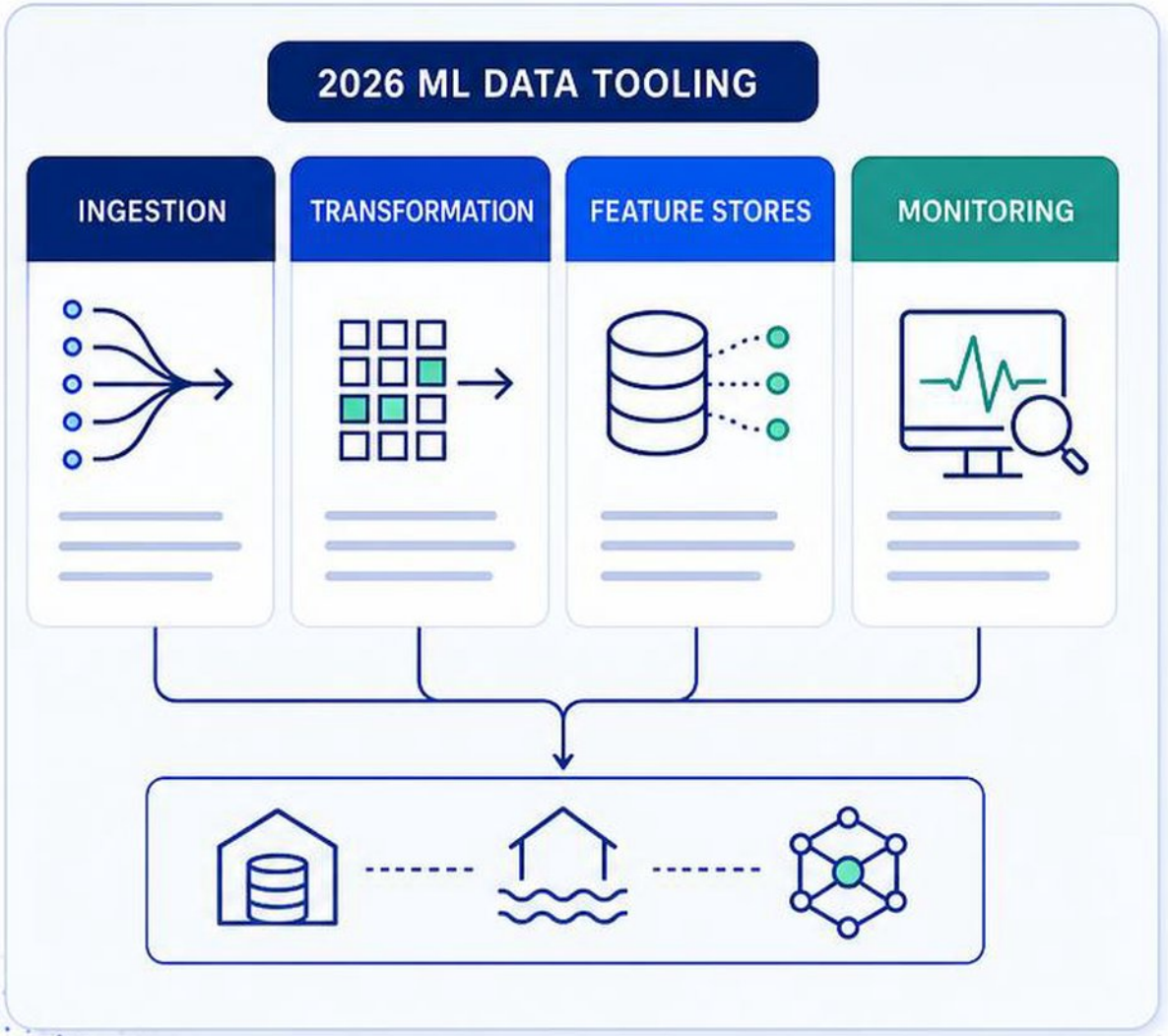
# Operationalizing and Monitoring ML Data

- Track freshness, completeness, schema validity, and distribution drift
- Detect schema skew, feature skew, and training-serving skew early
- Monitor null rates, row counts, and unseen category values
- Alert on feature staleness, serving latency, and pipeline failures
- Prepare incident runbooks and rollback for data-dependent ML

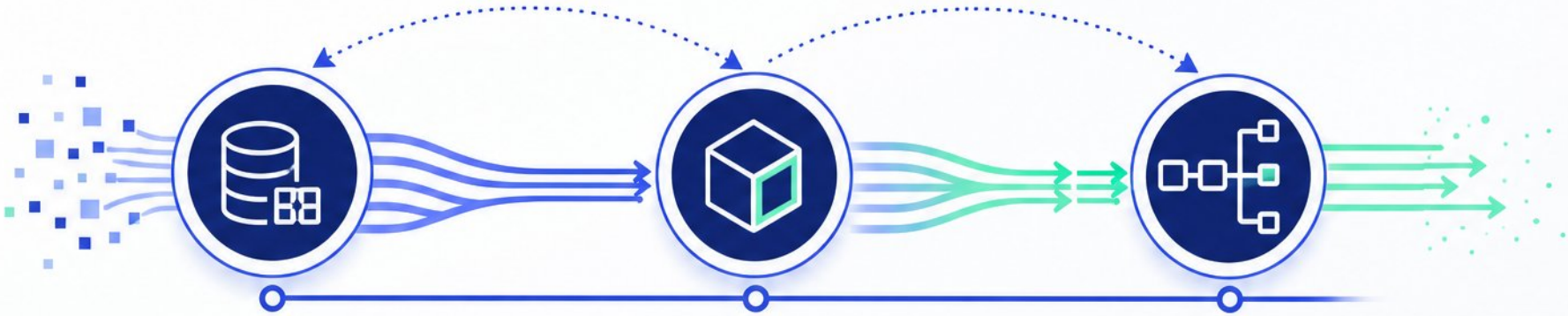


# Tools and Platform Selection

- Survey 2026 tooling: ingestion, transformation, feature stores, monitoring
- Compare hyperscalers, Databricks, and open-source/ self-hosted stacks
- Align choices with team skills, cloud footprint, and governance needs
- Evaluate integration across warehouse, lakehouse, and experimentation environments
- Prioritize data-ML proximity to reduce friction and improve governance



# Case Studies and Practical Patterns



**CENTRALIZED FEATURE STORES**
Unify and standardize data for reuse

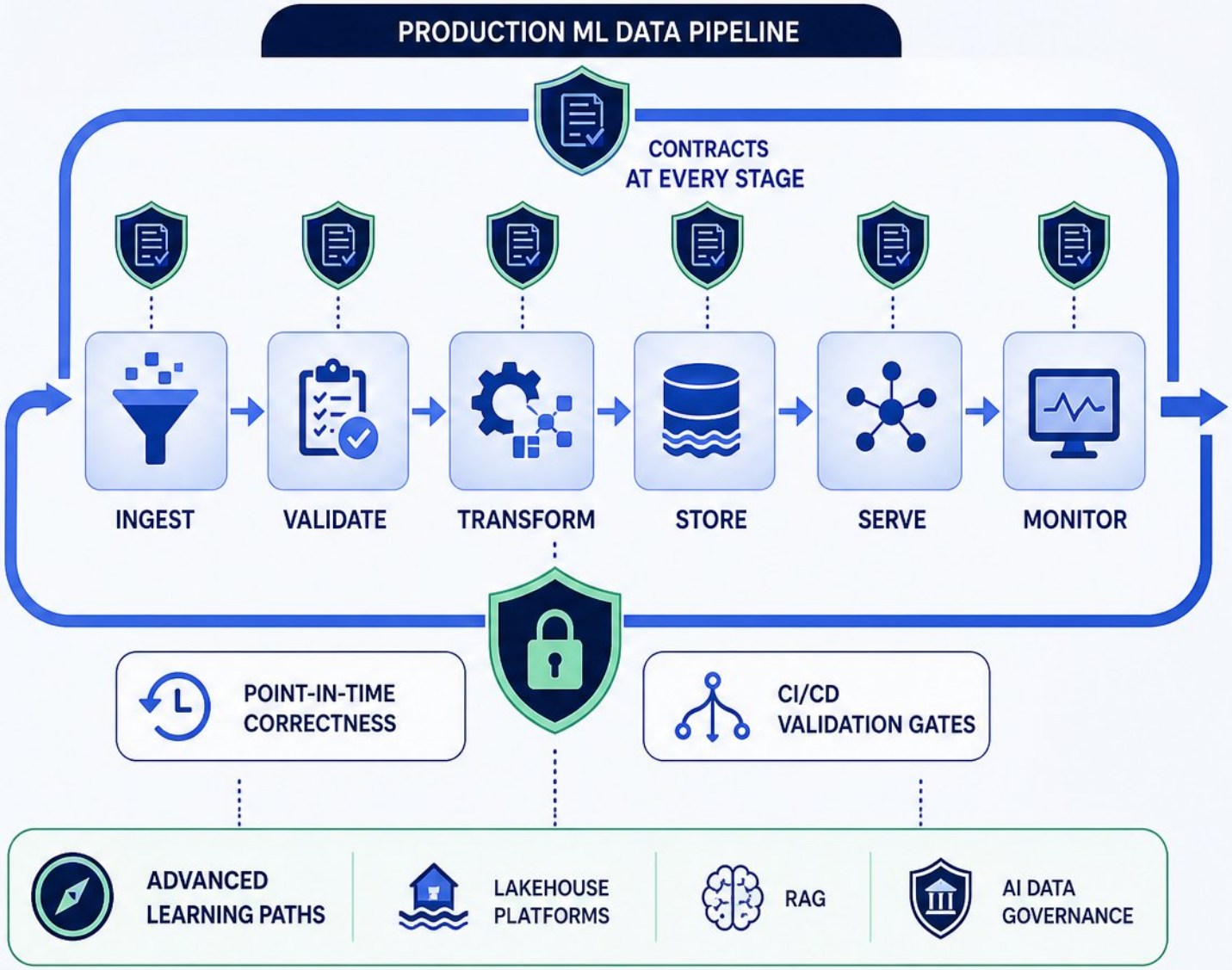
**STANDARDIZED DATA PRODUCTS**
Reliable, governed, and discoverable

**MODULAR PIPELINES**
Flexible, scalable, and adaptable to change

- Oportun consolidated to one platform, cutting ingestion from two weeks to two days
- BBVA gained 20-75% faster development with standardized MLOps and governance
- Uber fixed GPU starvation, reducing training time from 22 hours to 3
- Yelp's streaming lakehouse cut analytics latency from 18 hours to minutes
- Separate workload classes: batch, streaming, and inference need distinct patterns

# Next Steps and Learning Path

- Implement the full reference workflow: ingest, validate, transform, store, serve, monitor
- Standardize data contracts and feature stores to prevent skewed, silent failures
- Adopt team safeguards like point-in-time correctness and CI/CD validation gates
- Pursue advanced paths in lakehouse platforms, RAG, and AI data governance



PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/2eceaba4/public