MERIT

AI-Ready Data Infrastructure An Engineering and Architecture Guide for Technical Leaders

The engineering decisions across ingestion, storage, transformation, orchestration, serving, and governance that determine whether AI workloads reach production and perform reliably at scale.

For: VPs of Engineering | Data Engineers and Architects | Heads of Data Platform
Heads of Data Engineering | CTOs with hands-on platform accountability

www.meritdata-tech.com

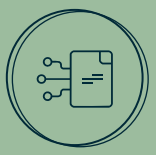
What's inside

Market context and engineering stakes: why data nfrastructure is the primary constraint on AI delivery in 2026

Infrastructure anti-patterns that block AI from reaching production - and the architectural fixes at each layer

Ingestion, storage, transformation, orchestration, & serving: engineering decisions, trade-offs, and failure modes at each layer

Governance as engineering: automated lineage, data contracts in CI/CD, and column-level access enforcement in production



Platform patterns for AI-ready data products: feature consistency, point-in-time correctness, and offline-online parity

Three production deployments: energy, maritime, and construction - technical approach and outcomes

A technical readiness diagnostic: seven questions against your current production stack

A note before you read this

This guide is written for engineering leaders who are already investing in data infrastructure -- and who are trying to understand why their AI initiatives are not reaching production at the rate they should be. It is not an introduction to data engineering concepts or a vendor evaluation guide.

Where we go deep into specific tools, patterns, and failure modes, there is a reason for it. Where we stay at the architectural and decision level, that is intentional too. The guide is structured around the decisions that matter -- not a comprehensive implementation manual for any single layer of the stack.

MERIT

The Data Engineering Imperative in 2026

Data engineering is no longer an infrastructure support function. It is the foundational layer on which every AI and ML initiative in your organisation either succeeds or fails. The market has recognised this -- the global big data and data engineering services market was valued at $75.55 billion in 2024 and is projected to reach $325 billion by 2033, growing at 17.6% CAGR. (MarketDataForecast, 2024)

That growth is not driven by a general interest in better pipelines. It is driven by the recognition that AI and ML initiatives are only as reliable as the infrastructure feeding them -- and that infrastructure, in most organisations, is where the constraint actually sits.

The failure rate tells you something important about how the industry is actually performing against that dependency.

The production gap

48% **of AI projects make it past pilot on average, and it takes 8 months to go from AI prototype to production** (Gartner, 2024)

43% **of organisations cite data quality and readiness as their primary obstacle to AI success -- ranking above model accuracy, computing costs, and talent** (Informatica CDO Insights, 2025)

60% **of AI projects unsupported by AI-ready data will be abandoned through 2026** (Gartner, 2025)

80% **of a data scientist's time is spent preparing data rather than building models**

The pattern is consistent. Teams invest in model development, MLOps tooling and AI talent, then discover that the pipeline feeding their models is the constraint. Features are computed inconsistently between training and serving. Schema changes in upstream systems break models silently. Data quality issues surface in production outputs rather than being caught at ingestion. The model gets blamed. The root cause is the infrastructure.

Model performance in production is a function of data infrastructure reliability. The teams shipping AI at the rate they need are the ones that treat pipeline quality, schema governance, and feature consistency as first-class engineering problems -- not operational afterthoughts

What this means for engineering teams in 2026

The organisations pulling ahead are those that have redesigned their data infrastructure around activation rather than storage -- around the latency, quality and governance requirements of AI workloads, not just the throughput requirements of analytics. The engineering decisions that create this advantage are not exotic. They are specific, addressable, and most of them can be resolved without a full re-platform.

The rest of this guide walks through exactly where those decisions sit, what good looks like at each layer, and how Merit has implemented them in production across multiple industries.

MERIT

Five Infrastructure Anti-Patterns That Kill AI in Production

These are not hypothetical failure modes. They are the patterns Merit encounters consistently when engaged to diagnose why an AI programme has stalled. Each one is fixable. Each one is also invisible until it causes a problem.

Anti-pattern 01 - The all-batch pipeline

The infrastructure anti-patterns in this chapter matter specifically because of how AI workloads differ from analytics workloads. Analytics can tolerate stale data. AI systems cannot -- not because freshness is always a real-time requirement, but because the gap between the data a model was trained on and the data it receives at inference time directly determines whether the model's predictions are valid. Training-serving consistency, inference latency, and the ability to detect data drift in production all depend on how the pipeline is designed. Getting the pipeline wrong does not produce a slow dashboard. It produces a model that degrades silently in production.

Scheduled batch jobs -- nightly Airflow DAGs running Spark transforms, daily warehouse refreshes via dbt, hourly Glue crawlers -- are the backbone of most data platforms built before 2022. For reporting workloads and many AI use cases operating on daily or weekly refresh cycles, they are perfectly adequate. The problem is not batch itself. It is a mismatch between the freshness guarantees a pipeline provides and the freshness requirements of the AI workload consuming it. A fraud detection model fed by a pipeline with 8-hour batch latency is not a fraud detection model. A recommendation engine trained on yesterday's user behaviour is optimising for the wrong objective. But a churn prediction model running on weekly aggregates may be entirely appropriate for its use case. The question is whether the pipeline's latency characteristics are matched to the workload's requirements -- and in most organisations, they are not, because the pipeline was built for analytics and the AI workload was bolted on afterward.

The path forward is not to replace batch wholesale. It is to introduce a streaming layer for latency-sensitive data paths where the workload genuinely requires it, using Kafka or Kinesis as the event bus, Flink or Spark Structured Streaming for stateful computation, and a unified serving layer that can read from both batch and streaming sources via the Lambda or Kappa architecture pattern.

Where Kafka is introduced, partition design is a real engineering decision, not a configuration detail. Partition count determines the maximum parallelism available to downstream consumers -- a topic with 12 partitions can be consumed by at most 12 parallel consumers in a single consumer group. Setting partition count too low at topic creation is a common mistake because Kafka does not allow partition count to be reduced, and increasing it later breaks ordering guarantees for keyed messages. The right approach is to size partitions against the target consumer parallelism at peak load, not against current throughput. For a Flink job processing user events across a 12-node cluster, 12 partitions aligns consumer parallelism to cluster capacity. For a topic where message ordering per user is a correctness requirement -- as it typically is in fraud detection and recommendation pipelines -- the partition key must be the user identifier, not a random or round-robin assignment, to ensure all events for a given user are processed in order by the same consumer.

Anti-pattern 02 - Schema drift without contracts

Upstream source schemas change. This is inevitable. The question is whether those changes are caught before or after they break downstream models. In most architectures the answer is after, because there is no enforcement mechanism between producers and consumers.

It is important to distinguish between three types of upstream change that affect AI workloads differently. Schema compatibility changes -- adding, removing, or renaming fields -- are structural and can be caught by a schema registry with compatibility enforcement. Semantic changes -- a field that still exists and still passes validation but whose business meaning has changed, such as a status field that previously contained three values and now contains seven -- are invisible to a schema registry entirely. Data quality regressions -- a field that is structurally valid but whose value distribution has shifted significantly, such as a transaction amount field where the mean has doubled due to a currency change upstream -- are also invisible to schema enforcement. A schema registry protects structure. It does not protect business meaning or statistical consistency. Contracts that cover only schema compatibility will miss the majority of upstream changes that degrade model performance in production.

Schema registries with compatibility enforcement address the structural layer. Apache Avro with the Confluent Schema Registry or AWS Glue Schema Registry enforces compatibility at the producer side before a message is accepted. BACKWARD compatibility allows existing consumers to read new data. FULL compatibility allows both old & new consumers to read both old and new data. Most event topics feeding ML feature pipelines should be FULL-compatible to avoid silent degradation during rolling deployments where old and new consumer versions coexist.

The subtlety that schema compatibility misses is worth illustrating. Consider a transaction event schema where a merchant category field is added as a nullable string with a default of null. Adding a nullable field with a default is a BACKWARD-compatible change -- existing consumers can read new messages without modification and it passes schema validation. But if a downstream feature pipeline is computing a spend-by-category aggregate that treats null as an unknown category, and the upstream team begins populating that field with a new taxonomy that maps differently to the categories the model was trained on, the feature values shift without any schema violation being raised. The model sees different feature distributions than it was trained on. Performance degrades. The schema registry reports no errors.

This is why data contracts need to cover semantic expectations and statistical quality thresholds alongside structural compatibility. A contract that specifies the expected cardinality of a category field, the maximum acceptable null rate, and the distribution of values against a baseline is a testable specification that catches the change the schema registry cannot. Tools like Soda Core and Great Expectations support contract-style quality definitions that can be enforced as pipeline gates at both the producer and consumer side.

Anti-pattern 03 - The unstructured data lake

Object storage -- S3, GCS, Azure Blob -- is cheap and scalable. It is not, by default, reliable as a data platform. The problem is not any specific file format. It is unmanaged object storage without transactional table semantics, versioning, or governance. Whether the files are Parquet, ORC, Avro, or JSON, a data lake without a table format layer has no ACID guarantees, no schema enforcement, and no safe concurrent write semantics. A failed write can leave partial data. Concurrent readers and writers can produce inconsistent results. There is no efficient mechanism for time-travel to a previous dataset version, which means reproducible training datasets are not possible. For AI workloads that require point-in-time correctness, consistent reads during model training, and the ability to audit what data a model was trained on, raw object storage is not a foundation -- it is a liability.

The solution is a table format layer on top of object storage. Delta Lake, Apache Iceberg, and Apache Hudi all implement transactional semantics in this way, but the choice between them is a design trade-off determined by workload profile, engine environment, update patterns, and table maintenance requirements -- not a categorical preference for one format over another.

Ecosystem fit is the first consideration. Teams running predominantly on Databricks and Spark will find Delta Lake's native integration reduces operational overhead significantly. Teams consuming the same feature tables across multiple engines -- Spark, Trino, Flink, Hive, Athena -- will find Iceberg's broad multi-engine compatibility reduces the risk of engine-specific read inconsistencies. Update patterns are the second consideration. Upsert-heavy pipelines merging CDC events into slowly changing dimensions favour Hudi's Merge-on-Read, which minimises write amplification in ways Delta and Iceberg do not match as efficiently. Append-heavy event tables with infrequent updates are well served by either Delta or Iceberg. Table maintenance is the third consideration. All three formats require compaction, snapshot cleanup, and retention management as ongoing operational tasks. The tooling and automation available for each differs by platform, and underestimating this operational burden is a common mistake when selecting a table format for production AI workloads.

Iceberg's hidden partitioning and partition evolution are useful properties for feature tables that change over time -- they allow partition strategy changes without data rewrites or query updates. But those properties only deliver value if Iceberg fits the broader engine and operational context. The workload profile determines the right format.

Anti-pattern 04 - Feature recomputation at serving time

Training-serving skew is the gap between the feature values a model was trained on and the feature values it receives at inference time. It is one of the most common causes of model performance degradation in production. The root cause is not the absence of a specific tool -- it is the absence of shared feature definitions and reproducible point-in-time logic across training and serving contexts. When feature computation logic is implemented separately in two places, those implementations drift. A mean-normalisation that uses the training set distribution in a batch pipeline will produce different values from one computed over a sliding 24-hour window in a serving pipeline. The model sees different feature distributions at training and inference time and behaves accordingly. The degradation is often silent -- the model continues to produce outputs, but their reliability against the original evaluation metrics no longer holds.

The solution is to establish a single authoritative feature definition that is computed consistently and served to both offline training jobs and online inference endpoints. How that is implemented depends on the engineering context. Some teams solve this through disciplined shared libraries and versioned feature computation code deployed across both training and serving environments, without adopting a dedicated feature store product. Others implement it through a centralised feature platform. Both approaches are valid if they enforce the same feature logic at training and serving time and provide reproducible point-in-time lookups.

Where a dedicated feature store is appropriate, the selection criteria matter more than the product name. Offline-online consistency is the primary requirement -- the feature store must guarantee that the same feature value is produced for a given entity at a given point in time regardless of whether it is being retrieved for model training or live inference. Point-in-time join correctness is critical for training dataset construction -- the feature store must be able to reconstruct the exact feature values that would have been available at the moment each training label was generated, without leaking future data into the training set. TTL handling determines how long feature values remain valid for online serving -- a feature with a 7-day TTL should not be served stale beyond that window, and the store must handle expiry consistently across offline and online paths. Registry management enables safe versioning and rollback of feature definitions -- when a feature computation changes, the registry must track which model versions were trained against which feature versions. Serving latency for online retrieval must meet the inference SLA of the consuming model -- a feature store that cannot serve values within the latency budget of a real-time inference endpoint is not fit for that use case regardless of its other capabilities.

Feast, Tecton, AWS SageMaker Feature Store, and Databricks Feature Store each address these requirements to different degrees and with different operational trade-offs across cloud environments, engine compatibility, and infrastructure complexity. Evaluating them against the five criteria above -- consistency, point-in-time correctness, TTL handling, registry management, and serving latency -- against your specific workload is the right framing, not a general comparison of product features.

Anti-pattern 05 - Governance as documentation, not as code

Most data governance implementations conflate descriptive governance with enforcement.
A Confluence page listing data owners, a spreadsheet tracking sensitive fields, a data catalogue storing metadata -- these are not without value. Descriptive governance establishes accountability, communicates intent, and provides context that automated systems cannot infer. The problem is when descriptive governance is the only governance. Documentation does not prevent a training job from ingesting PII. A data catalogue does not stop a schema change from reaching production. A spreadsheet of sensitive fields does not tell you, when a model produces a biased output, which version of which dataset it was trained on. The failure mode is governance without enforcement -- where the policies exist in documentation but are not implemented as controls in the system itself.

Governance as engineering requires three distinct categories of control, each addressing a different concern.

- **Lineage capture** is about reproducibility and root-cause traceability. OpenLineage -- the CNCF standard for lineage metadata -- instruments existing orchestration and transformation tools to emit lineage events automatically. The Airflow OpenLineage provider emits events for every task that reads or writes data. The dbt OpenLineage integration emits lineage from the dbt manifest. These events are collected by a backend -- Marquez for open source deployments, or commercial catalogues like DataHub, Atlan, or Collibra -- and made queryable as a graph. The practical output is the ability to answer, for any given model output, what data it was trained on, what transformations were applied, and whether anything in that chain has changed. Lineage coverage is only as strong as instrumentation depth -- adopting OpenLineage does not automatically provide complete traceability across all pipelines. Every orchestration layer, transformation tool, and ingestion path needs to be instrumented for the lineage graph to be meaningful.

- **Access enforcement** is about preventing unauthorised data consumption at the compute layer, not the application layer. Column-level masking policies in Snowflake, column-level access controls in BigQuery, and row and column filters in Databricks Unity Catalog apply access rules automatically at query time. No application needs to enforce them because the compute layer does. This is the only access enforcement model that scales reliably -- application-level controls break when data moves to a new consumer, which AI workloads require constantly. Table-level access is insufficient for AI. A training job with read access to a customer table has access to every PII field in that table. Column-level enforcement is the control that prevents sensitive fields from entering model training pipelines without explicit authorisation.

- **Sensitive data discovery** is about ensuring that PII classification is a systematic process rather than a human memory exercise. Microsoft Presidio -- open source -- and AWS Macie -- managed -- can classify PII fields automatically across structured datasets. The output feeds a tagging system -- Apache Atlas entity classifications or Databricks Unity Catalog tags -- that drives the column masking policies described above. When a new dataset is onboarded, PII classification runs as a step in the ingestion pipeline. Sensitive field identification is not dependent on a person remembering to tag a new dataset before it reaches a model training job. The classification is also not infallible -- automated scanners have false negative rates that vary by data type and domain, so classification output should be treated as a first pass that requires periodic validation rather than a complete and final inventory.

  Descriptive governance -- catalogues, data dictionaries, ownership registries -- remains valuable as the layer that communicates context and intent. The distinction is that it should complement programmatic enforcement, not substitute for it.

The AI-Ready Stack: Layer by Layer

What follows is a reference architecture for an AI-ready data stack, structured across five layers. It is not a universal blueprint -- the right choices at each layer depend on your workload profile, engine environment, team expertise, and existing platform investments. Where we name specific tools and patterns, we explain the reasoning and the trade-offs. The goal is to give engineering leaders a structured frame for evaluating decisions at each layer, not a prescribed stack to adopt wholesale.

Layer 01 - Ingestion: unified batch and streaming

The ingestion layer has three modes and all three need to be first-class: batch ingestion for historical loads and cost-sensitive workloads (AWS Glue, Azure Data Factory, Fivetran, Airbyte), streaming ingestion for real-time event data (Kafka with schema registry, Kinesis, Pub/Sub), and change data capture for operational database replication (Debezium on Postgres/MySQL, AWS DMS for managed CDC).

The ingestion layer should not contain business logic -- but that is different from saying it should do no transformation at all. Envelope unwrapping, field normalisation, PII redaction, and quality rejection are all appropriate at ingestion. What should not live here is derived field computation, aggregation, or any transformation that encodes domain knowledge. Business logic at ingestion is untested, unversioned, and invisible to the lineage graph.

Dead letter queues on every streaming topic are a baseline requirement, but the DLQ is only useful if there is an operational pattern behind it. Alerting thresholds need to be defined so that DLQ depth triggers action rather than accumulating silently. A root-cause workflow needs to exist so that when a DLQ alert fires, the team knows whether the failure is a schema violation, a processing error, or a producer-side change -- each has a different resolution path. And a replay strategy needs to be tested before it is needed -- covering ordering guarantees for keyed topics, idempotency in downstream consumers, and time-window alignment for feature computation. A DLQ without a tested replay path is a dead end, not a recovery mechanism.

Layer 02 - Storage: lakehouse over raw lake

Object storage as the foundation. Delta Lake, Iceberg, or Hudi on top for transactional semantics. The choice between them is a design trade-off determined by workload profile, engine environment, update patterns, and table maintenance requirements -- no single format is the correct default for AI workloads. Delta Lake integrates most deeply with Databricks and Spark environments and provides strong write consistency with operational simplicity for Spark-native teams. Iceberg offers the strongest multi-engine compatibility across Spark, Trino, Flink, Hive, and Athena, and its time-travel and partition evolution capabilities are worth evaluating for feature tables that change over time -- but only where Iceberg fits the broader engine and operational context. Hudi is best suited to upsert-heavy pipelines merging CDC events into slowly changing dimensions, where its Merge-on-Read storage type minimises write amplification. Evaluate against your specific engine investments, consumer access patterns, and operational context before committing to a table format.

Day-two operations are where most production lakehouses encounter problems that were not planned for at build time. Four operational concerns recur consistently.

- **Compaction.** High-frequency streaming writes without compaction produce small file proliferation -- one of the most common causes of query performance degradation in production lakehouses. Compaction jobs rewrite small files into larger ones and update table statistics. The right target file size depends on the query engine, the query pattern, and storage I/O characteristics -- there is no universally correct range. The principle is that files should be large enough to minimise metadata overhead and small enough to support partition pruning for the query patterns your workloads actually run. Compaction schedules set once and never revisited as data volumes grow are a consistent operational failure mode.

- **Snapshot management and retention.** Every write to a Delta, Iceberg, or Hudi table creates a new snapshot. Without a retention policy, snapshot history accumulates indefinitely, increasing metadata overhead and storage costs. Retention policies need to balance two competing requirements: enough history for time-travel queries supporting model reproducibility and audit, and aggressive enough cleanup to prevent metadata bloat from degrading query planning performance. Both the retention window and the cleanup schedule need to be defined explicitly and monitored.

- **Metadata operations.** As tables grow, metadata files -- transaction logs, manifest lists, statistics files -- can become a performance bottleneck in their own right. Periodic metadata compaction, statistics refresh, and orphan file cleanup are maintenance tasks that most teams discover the hard way when query planning times begin to degrade. These operations need to be scheduled, monitored, and tuned as part of routine platform maintenance, not treated as one-off remediation tasks.

- **Schema evolution.** All three table formats support schema evolution, but the operational discipline around it matters. Schema changes should go through the same review and validation process as code changes -- with downstream impact assessment, consumer notification, and compatibility verification before the change lands in production.

Layer 03 - Transformation: dbt as the standard, not an option

Many organisations run transformation on Spark, Flink SQL, Dataform, native warehouse transformation engines, or mixed approaches -- and for good reason. The right transformation layer depends on the workload, the team's expertise, and the existing platform. Where SQL-led warehouse transformation is the dominant pattern, dbt is a strong standardisation choice. The engineering case for it is specific. dbt models are SQL files in a git repository, which means they are version-controlled, code-reviewed, and deployable via CI/CD in the same way as application code. dbt tests -- not_null, unique, accepted_values, relationships -- run at build time, which means bad data is caught before it reaches downstream consumers rather than surfacing in model outputs. dbt docs generate a live data dictionary from the model definitions, which means documentation stays in sync with the code rather than drifting in a separate wiki. dbt lineage is exposed via the manifest artifact, which feeds directly into OpenLineage-compatible catalogues. These are engineering properties, not product marketing claims -- and they are the reason dbt is worth standardising on where the SQL-led pattern fits.

One structuring model that works well for AI workloads is the medallion architecture: raw models that reflect source data exactly with no transformation, staging models that standardise, clean, and type-cast, and mart models that apply business logic and aggregate for specific consumers. Feature tables for ML built as mart-layer models are tested, documented, and versioned like any other piece of production infrastructure rather than maintained as ad-hoc queries. However, medallion is one approach among several. Data-product-oriented models that organise transformation around domain ownership, or domain-driven structures that align with the team boundaries in a data mesh, are equally valid depending on the organisational context. The structuring model should reflect how your teams own and consume data, not a universal prescription about how transformation layers should be named.

Layer 04 - Orchestration: observable pipelines, not just scheduled ones

The gap between an orchestrator used as a cron replacement and one used as an observable data platform is significant -- and for AI workloads, that gap is consequential. A pipeline that runs on a schedule but provides no visibility into what it processed, what it skipped, and what state it left downstream systems in is not production infrastructure. It is a black box that produces outputs until it silently does not.

- **Observability** is the first engineering requirement. Every pipeline task needs to emit structured metadata -- records processed, records rejected, quality gate results, processing latency, and downstream dependency status -- not just a success or failure exit code. This metadata needs to be queryable, not just visible in a UI. When a model produces unexpected outputs, the first question is always what data it consumed and whether anything in the pipeline behaved differently from the previous run. That question needs to be answerable in minutes, not hours. Airflow 2.x, Prefect, and Dagster all support this to varying degrees -- the decision criteria are how deeply observability is native to the framework versus bolted on, and whether the metadata produced integrates with your lineage backend.
- **Retry and failure handling** needs to be designed per task, not set uniformly across the pipeline. A transient network failure on an API call warrants an exponential backoff retry. A data quality gate failure should not retry at all -- it should fail fast, alert, and halt downstream tasks until the root cause is resolved. Retrying a quality gate failure propagates bad data further downstream on each attempt. The retry policy is a correctness decision, not just an availability decision.

- **Backfill strategy** is a first-class design concern for AI pipelines, not an edge case. When a pipeline fails mid-run or is modified, the ability to reprocess a specific time window without affecting other windows -- and without duplicating records that were already processed -- depends on idempotent task design from the outset. Tasks that are not idempotent make backfills dangerous. Incremental processing logic, watermark management, and partition-aligned task boundaries need to be designed in before the first production run, not retrofitted after the first backfill incident.
- **State management** matters specifically for stateful streaming and incremental batch pipelines. The orchestrator needs to track what has been processed, what watermark the pipeline last reached, and what state needs to be checkpointed for recovery. For Flink-based streaming pipelines, checkpointing intervals and state backend selection -- RocksDB for large state, in-memory for low-latency small state -- are engineering decisions that determine recovery time objectives. For incremental dbt models or Spark batch jobs, the high-water mark needs to be persisted and recoverable independently of the orchestrator's own metadata store.
- **Dependency control** determines whether a pipeline can proceed safely when upstream data has not arrived, is late, or has arrived but failed quality checks. Dataset sensors, SLA miss alerts, and explicit upstream dependency declarations prevent downstream tasks from running on incomplete inputs. For AI feature pipelines specifically, a training job that runs against a feature table that was not fully refreshed due to a silent upstream failure will produce a model trained on stale or incomplete data -- with no indication at training time that anything was wrong.

  Data quality checks need to be first-class citizens in the pipeline definition itself, not separate monitoring jobs. A quality gate that can fail the pipeline is a control. A quality dashboard that reports after the fact is an observation. For AI workloads, the distinction is not semantic -- bad data that reaches a model training job has already caused the problem by the time a dashboard flags it.

Layer 05 - Serving: consistency between training and inference

The serving layer is the most neglected layer in most data stacks and the most consequential for AI. It needs to address four engineering concerns that are frequently underspecified at build time.

- **Online store latency and feature freshness SLAs.** The online feature store serving live inference endpoints has a latency budget determined by the inference SLA of the consuming model. A fraud detection model with a 50ms response requirement cannot tolerate a feature store that takes 30ms to retrieve a feature vector -- that leaves no headroom for model inference itself. Online store latency needs to be measured, baselined, and monitored as a first-class SLA, not treated as an infrastructure detail. Alongside latency, feature freshness SLAs need to be defined per feature -- the maximum acceptable age of a feature value at serving time before it is considered stale. A feature with a 7-day TTL serving a model that was trained on daily refresh cycles is operating within its freshness contract. The same feature serving a real-time recommendation model may not be. Freshness SLAs need to be explicit, enforced at the store level, and monitored in production.

- **Model registry and versioned artefacts.** A model registry -- MLflow, SageMaker Model Registry, Vertex AI Model Registry -- provides versioned model artefacts with associated metadata: training dataset version, feature store snapshot, hyperparameters, and evaluation metrics. This metadata is not administrative overhead. It is the information required to diagnose production degradation, reproduce a training run, and execute a safe rollback. A model in production without a registry entry linking it to its training data and feature versions cannot be debugged or rolled back reliably.

- **Model rollback**. Rollback capability needs to be designed before it is needed, not improvised after a production incident. A safe rollback requires that the previous model version is still registered and its serving infrastructure is still available, that the feature versions it was trained on are still accessible in the online store, and that traffic can be shifted back to the previous version without a redeployment cycle. Registry management, feature store retention policies, and deployment infrastructure all need to be designed with rollback as an explicit requirement.

- **Shadow testing observability.** Blue-green and canary deployment patterns reduce the risk of model updates by controlling the blast radius of a bad model reaching production. But the risk reduction only materialises if the shadow testing phase is instrumented to produce actionable signal. Running a new model version against live traffic in shadow mode -- without affecting outputs -- is only useful if the shadow outputs are captured, compared against the production model's outputs on the same inputs, and evaluated against defined divergence thresholds before cutover proceeds. Shadow testing without observability is a ceremonial step. With observability -- divergence metrics, distribution comparisons, latency profiles, and error rates captured and queryable -- it is the control that makes safe model deployment possible at the pace AI teams need to ship.

MERIT

Governance Embedded in the Pipeline

Governance that lives outside the pipeline does not govern AI systems. The AI-specific risks that make this consequential are concrete: a model trained on data whose provenance cannot be reconstructed cannot be audited or defended in a regulated environment; a model trained on sensitive fields it should not have accessed may encode spurious correlations that are invisible until they cause a compliance or quality failure; and a model deployed without version-linked training data cannot be safely rolled back when its behaviour changes in production. These are not data governance problems in the abstract -- they are AI deployment risks that require programmatic controls in the pipeline itself.

Automated lineage with OpenLineage

OpenLineage instruments existing orchestration and transformation tools to emit lineage events automatically. The Airflow provider emits events for every task that reads or writes data. The dbt integration emits lineage from the manifest. Spark integration captures job-level lineage from execution plans. These events are collected by a backend -- Marquez for open source deployments, or DataHub, Atlan, or Collibra commercially -- and made queryable as a graph.

The practical output is traceability: for any model output, you can identify the upstream datasets, the transformations applied, and whether anything in that chain has changed. When a source schema changes, every downstream AI model affected can be identified immediately. When a model degrades in production, the training data version and transformation history can be traced in minutes rather than days.

Lineage coverage is only as strong as instrumentation depth and metadata consistency. Adopting OpenLineage does not automatically produce complete traceability. Every orchestration layer, transformation tool, and ingestion path needs to be instrumented. Gaps in instrumentation produce gaps in the lineage graph -- and those gaps are typically invisible until a debugging or audit scenario surfaces them

Data contracts in CI/CD

Data contracts are formal agreements between producers and consumers, but schema, freshness, and statistical quality are distinct contract types with different enforcement approaches and different failure consequences for AI workloads.

Schema contracts specify structural expectations -- field names, types, nullability, and compatibility mode. These are enforced at the producer side via a schema registry before a message is accepted or a table write is committed. A schema contract violation is a hard failure that should halt the pipeline.

Freshness contracts specify the maximum acceptable age of data at consumption time -- the event timestamp must be within the last 24 hours, or the feature table must have been refreshed within the last 6 hours. These are enforced as pipeline gates at ingestion or at the feature serving layer. A freshness violation means a model may be consuming stale inputs, which affects inference correctness rather than structural validity.

Statistical quality contracts specify distributional expectations -- the distribution of a categorical field must not deviate more than a defined threshold from baseline, the null rate of a field must not exceed a defined percentage, the mean of a numeric field must remain within a defined range. These are enforced via tools like Soda Core or Great Expectations run as pipeline tasks. A statistical violation may indicate an upstream semantic change that passes schema validation but breaks feature derivation logic.

All three contract types need to be automated. Producer-side tests run before data is published. Consumer-side tests run at ingestion before data enters the transformation layer. Neither should rely on a human reviewing a report after the fact.

Column-level security and PII classification

Sensitive and unnecessary fields should be excluded or masked from AI training pipelines according to purpose, legal basis, and policy -- not only because they create regulatory exposure, but because fields that are irrelevant to the modelling objective introduce noise, encode spurious correlations, and create lineage obligations that most organisations cannot satisfy at audit time. Data minimisation is a design principle applied at the pipeline level, not a compliance checkbox applied at the point of data request.

Automated classification is the only approach that scales. Microsoft Presidio and AWS Macie can classify sensitive fields across structured datasets as part of the ingestion pipeline. The output feeds a tagging system -- Apache Atlas or Databricks Unity Catalog tags -- that drives column masking policies enforced at query time in Snowflake, BigQuery, or Databricks Unity Catalog. When a new dataset is onboarded, classification runs as a pipeline step. Sensitive field identification is not dependent on a person remembering to tag a dataset before it reaches a training job. Classification output should be treated as a first pass requiring periodic validation -- automated scanners have false negative rates that vary by data type and domain.

MERIT

How Merit Architects the Intelligence Layer

Merit's data engineering engagements are structured around five delivery workstreams that cover the full journey from assessment to operational handover.

Architecture assessment. We start with your actual stack -- mapping data flows, integration dependencies, pipeline latency profiles, governance gaps, and the specific constraints blocking your AI workloads from reaching production. The output is not a generic maturity scorecard. It is a ranked inventory of the engineering decisions that need to be made, in the order they need to be made, with the trade-offs at each decision point documented against your existing investments and team capabilities.

Platform build. We design and build the target data platform with your engineers -- ingestion layer, lakehouse storage, transformation pipeline, orchestration, and serving infrastructure. Technology choices are made against your workload profile, engine environment, and operational context. We write the code, configure the infrastructure, and deploy through your CI/CD pipelines. Every component is built to be operated by your team without dependency on Merit once the engagement closes.

Observability. We instrument every layer of the stack for operational visibility -- pipeline health, data quality gate results, feature freshness SLAs, lineage coverage, and model serving metrics. Alerting thresholds, on-call runbooks, and root-cause workflows are defined and tested as part of the build, not left for the team to establish after go-live.

Handover. Every engagement closes with your team in a position to operate, extend, and own what has been built. That means architectural decision records documenting why choices were made, runbooks covering operational procedures and failure recovery, and hands-on knowledge transfer throughout the programme rather than a documentation pack at the end.

What we bring to an engagement

- **Platform engineering.** Cloud-native data platform design and build across AWS (Glue, Athena, EMR, Kinesis, S3, Lake Formation, SageMaker), GCP (BigQuery, Dataflow, Pub/Sub, Vertex AI), and Azure (Synapse, Data Factory, Event Hubs, Azure ML). Lakehouse implementation on Delta Lake, Apache Iceberg, and Apache Hudi -- covering ACID transactions, schema evolution, time-travel, compaction, and partition management. Workflow orchestration via Apache Airflow 2.x, Prefect, and Dagster -- with observable, self-healing pipelines, data quality gates, retry handling, backfill management, and dependency control built in.

- **Streaming and distributed compute.** Apache Spark for batch and structured streaming workloads. Apache Flink for stateful stream processing. Kafka for event streaming with schema registry and compatibility enforcement. Transformation and testing via dbt Core and dbt Cloud with full test coverage, incremental model patterns, and OpenLineage integration.
- **Governance and compliance.** Automated lineage via OpenLineage with Marquez and DataHub backends. Contract testing via Soda Core and Great Expectations. Column-level masking and row-level security across Snowflake, BigQuery, and Databricks Unity Catalog. PII classification via Presidio and Macie. Audit-ready lineage for regulated environments. ISO 27001 certified.
- **MLOps and feature engineering.** Feature store implementation across Feast, Tecton, SageMaker Feature Store, and Databricks Feature Store for consistent training-serving pipelines. Model registry and versioning via MLflow and platform-native registries. Deployment patterns covering canary, blue-green, and shadow scoring.
- **Modernisation.** Legacy data infrastructure assessment and migration -- from on-premise Hive, HDFS, and warehouse environments to cloud-native lakehouse architectures. AI-assisted code conversion for legacy pipeline languages. CDC-based migration patterns that maintain operational continuity during transition. DevSecOps enablement covering CI/CD pipeline build, infrastructure as code, and automated testing frameworks for modernised data environments.

Deep engineering expertise across cloud, pipeline and governance. Sector-proven across multiple industry verticals. End-to-end ownership from strategy through to deployment.

How we work

We work directly in your stack. Your repositories, your cloud accounts, your CI/CD pipelines. We do not produce architecture diagrams and leave you to implement them. We write the code, run the tests, build the observability and hand over infrastructure that your team can operate, extend and own.

Every engagement follows the same engineering discipline: technical discovery against your actual stack, target architecture designed with your team, hands-on build with your engineers, validation against defined quality thresholds, and knowledge transfer that ends the engagement with your team more capable than when it started. Every engagement is structured so that your team can operate what has been built without ongoing reliance on Merit. That means pairing Merit engineers with your engineers throughout the build -- not delivering completed components for handover at the end. It means working in your repositories, following your coding standards, and building against your CI/CD pipelines so that the infrastructure is yours from the first commit. It means architectural decision records documenting why choices were made, runbooks covering operational procedures and failure recovery for every pipeline and platform component, and data quality and observability tooling that your team knows how to interpret and act on. The engagement closes when your team can extend, debug, and operate the platform independently -- not when the last component is deployed

Three Production Deployments

The following are from Merit's production data engineering work. Each is described at the technical level to give you an honest picture of what the work involved, not a marketing summary of what it achieved.

Global energy pricing intelligence: 35M records/day at 99.8% accuracy

A global media and intelligence company needed continuous pricing data capture from 800 global sources -- pricing assessments informing physical trade and financial markets in 100+ countries. Manual processes could not meet the volume, latency or accuracy requirements.

- **Technical approach:** Python-based scraping infrastructure with a resilient ETL framework and automated validation at every ingestion step. Dynamic scraper configuration to handle source format variation. Automated anomaly detection on incoming price data to catch outliers before they propagate downstream. Continuous 24/7 operation with self-healing retry logic and dead letter capture for failed extractions.
- **Outcome:** 35 million records processed per day at 99.8% accuracy. 30% reduction in operational costs. 50% faster market insights. Architecture built for 100% horizontal scalability without re-engineering as new sources are added.

Maritime intelligence cloud migration: 150TB, zero data loss, 70% fewer data issues

A UK-based maritime intelligence provider -- 500+ agents, 170 countries, 60+ daily analysts -- was running on on-premise Hive infrastructure that could not scale to meet growing analytical complexity. Maintenance costs were rising and pipeline latency was increasing.

- **Technical approach:** Multi-phase migration: one-time bulk migration of 150TB from on-premise Hive to AWS S3 with Parquet conversion and partition alignment. Incremental CDC feeds replacing manual extracts via automated scheduled pipelines. Spark-based ETL layer implemented on AWS Glue with the Glue Data Catalog for schema management. Athena-based dimensional data models enabling ad-hoc SQL analysis directly against S3. AWS SNS for real-time pipeline event notifications replacing manual status checks.
- **Outcome:** 150TB migrated with zero data loss. Analysis and reporting time reduced by 50%+ via Spark ETL replacing legacy batch processes. 70% reduction in client-reported data quality issues through automated quality reporting in the pipeline. Real-time process notifications replacing 6 to 12 hour delays.

Construction data aggregation: 475 councils, 48-hour turnaround, 50% cost reduction

A leading UK construction intelligence provider needed comprehensive planning application data -- applications, statements, reports and architectural documents, approximately 250 data points per project -- from 475 local councils across the UK and Ireland. Each council uses different portal software, different document formats and different data structures. Manual extraction was not viable at this scale.

- **Technical approach:** Scalable scraper fleet with per-council configuration for format variation, built to handle portal software differences across all 475 sources. Automated document parsing and structured data extraction from unstructured planning documents. Automated validation pipeline reducing manual quality checks by 70%. Incremental refresh architecture ensuring new planning applications are captured within the 48-hour SLA.
- **Outcome:** 40% richer data (more fields captured per project). 30% broader coverage (more councils reached). 50% cost savings versus manual extraction. 48-hour end-to-end turnaround, one day faster than the previous process.

Technical Readiness Diagnostic

Seven questions. Answer against your current production stack, not your roadmap. Enterprise data stacks rarely pass or fail these questions cleanly -- most organisations operate in hybrid states across multiple dimensions. The value is in identifying where partial implementations, coverage gaps, and architectural debt are concentrated, and what that means for your AI workloads specifically.

For each question, assess against three positions:

- Fully in place and consistent across the estate
- Partially in place with known gaps or exceptions;
  or
- Not in place.

- **Do you have a streaming ingestion path (Kafka, Kinesis, Pub/Sub) for your latency-sensitive data, or does everything go through batch ETL?**

  Fully in place means streaming ingestion is instrumented for all latency-sensitive data paths with defined freshness SLAs. Partially in place typically means streaming exists for some sources while others -- often older or more complex operational systems -- still rely on batch. Not in place means all data movement is batch-dependent regardless of downstream latency requirements. The partial state requires mapping: which AI workloads depend on which sources, and whether the sources still on batch are feeding time-sensitive inference pipelines. A churn model running on weekly aggregates may be unaffected. A fraud detection model fed by a batch pipeline is functionally constrained regardless of how good the model is

- **Is schema compatibility enforced at the producer side via a schema registry, or do you discover breaking changes when downstream jobs fail?**

  Fully in place means a schema registry with compatibility enforcement is in place across all event topics and data streams feeding AI workloads. Partially in place means registry enforcement exists for some topics but not others -- typically newer pipelines are covered while legacy sources are not. Not in place means schema changes are uncontrolled and breaking changes are discovered reactively. The partial state requires identifying which uncovered topics feed ML feature pipelines -- those are the ones where silent schema drift will eventually cause model degradation without any upstream alert.

- **Are your transformation models in dbt (or equivalent) with test coverage and version control, or do critical transforms live in ad-hoc notebooks or unversioned SQL?**

  Fully in place means all transformation logic is version-controlled, tested, and deployed through CI/CD pipelines. Partially in place -- the most common enterprise position -- means modern pipelines coexist with legacy stored procedures, ad-hoc notebooks, or unversioned SQL that predates the current team. Not in place means transformation logic is predominantly undocumented and undeployable safely. The partial state requires an inventory: which legacy transformations carry business-critical logic that feeds AI workloads, and which present the highest risk of silent drift between what the code does and what the business expects it to do.

- **Do you have a feature store serving consistent feature values to both training and inference pipelines, or are features recomputed independently in each context?**

  Fully in place means a single authoritative feature definition -- whether through a dedicated feature store or a shared feature library -- is used consistently across training and serving. Partially in place means some features are served consistently while others are recomputed independently in training and serving contexts, creating the conditions for training-serving skew in those pipelines. Not in place means feature computation logic is duplicated across training and serving with no consistency guarantee. The partial state requires identifying which independently recomputed features feed production models -- those are the ones where distribution drift between training and inference is most likely to be causing silent performance degradation.

- **Are data quality checks enforced as pipeline gates (failing the job on quality violations) or surfaced as post-hoc monitoring reports?**

  Fully in place means quality is enforced as a gate at ingestion, transformation, and serving -- violations fail the pipeline rather than propagating downstream. Partially in place means quality gates exist on some pipelines but not others -- typically newer pipelines are gated while legacy pipelines predate quality infrastructure. Not in place means quality is monitored after the fact, after data has already reached consumers. The partial state requires mapping which pipelines feeding AI workloads are ungated -- those are the ones carrying silent quality risk that surfaces in model outputs rather than in pipeline alerts.

- **Is data lineage tracked automatically across your pipelines (via OpenLineage or equivalent), or would tracing a model output back to its source data require manual investigation?**

  Fully in place means automated lineage is instrumented across all orchestration, transformation, and ingestion layers, with a queryable lineage graph covering all AI workloads. Partially in place means lineage is captured for some layers but not others -- a common state where dbt lineage is available but streaming pipelines or legacy batch jobs are not instrumented. Not in place means lineage is entirely manual. The partial state has a specific consequence for AI: the uninstrumented layers are exactly where root-cause investigation will stall when a model produces unexpected outputs in production.

- **Is access control enforced at the column level in your compute layer (Snowflake masking policies, BigQuery column-level access, Databricks Unity Catalog), or only at the table or database level?**

  Fully in place means column-level masking policies and row-level security are enforced automatically at query time across all datasets used in AI training and inference. Partially in place means column-level enforcement exists for some data domains or sensitivity tiers while others rely on table-level access or application-level controls. Not in place means access control is table-level or coarser throughout. The partial state requires identifying which datasets used in model training carry sensitive fields behind only table-level access -- those are the ones where a training job with legitimate table access may be ingesting fields it should not have access to.

Reading the results

A pattern of fully in place responses indicates a stack that can support AI workloads without resolving foundational blockers first. A pattern of partially in place responses -- the most common enterprise position -- indicates that sequencing decisions should be driven by where the gaps intersect with the AI use cases in scope. A pattern of not in place responses across multiple dimensions indicates that foundational data engineering work is a prerequisite before AI workloads can be expected to perform reliably in production.

The diagnostic is not a pass/fail threshold. It is a sequencing input. The gaps it surfaces are the constraints your AI programme is already working around -- whether or not they are currently visible in your metrics.

Ready to close the gap?

If your AI initiatives are not reaching production at the rate you need, or are degrading between evaluation and production, the root cause is almost certainly in the data infrastructure. It is a solvable problem - but it needs to be solved at the engineering level, with engineers who have solved it in production before.

Talk to Merit's data engineering team. We will assess your current stack honestly, identify the specific gaps between where you are and where your AI ambitions need you to be, and work alongside your team to close them.

About Merit

Merit Data & Technology, part of Merit Group Limited, is a trusted partner in AI-driven data and digital transformation. With over two decades of experience, We deliver scalable, secure and AI-ready automation and data solutions tailored to business needs. **Read More About Us**

www.meritdata-tech.com