MERIT

The CTO's Guide to Legacy Modernisation: Engineering AI-Ready Platforms

A practitioner's guide to migration sequencing, data architecture, and operational risk in enterprise legacy modernisation.

For: CTOs | VPs of Engineering | Enterprise Architects | Heads of Technology

www.meritdata-tech.com

What's inside

Why your legacy stack is the real reason your AI initiatives are stalling

The four technical debt patterns that block AI adoption at the infrastructure level

Modernisation roadmap design: sequencing, architecture decisions, and risk boundaries

AI-ready data architecture lakehouses, data mesh, and governance that actually works

How to modernise without stopping the business

Three production modernisations: AgTech, automotive, construction

A readiness diagnostic for your current stack

How Merit works as a modernisation partner

A note before you read this

This guide is structured around the decisions that determine whether a modernisation programme succeeds or stalls: how to sequence migration, map dependencies, design for coexistence, govern the transition, and modernise at the code level. These are not theoretical frameworks- they are drawn on Merit's internal engineering practice and the expertise of industry specialists embedded in our delivery teams.

CHAPTER 1

The Three Architectural Constraints That Block AI Delivery in Legacy Environments.

Most legacy modernisation programmes are framed as cost and risk decisions. In 2026, they are architecture decisions with direct AI consequences. Legacy systems impose three structural constraints on AI delivery: restricted governed data access, limited platform change velocity, and no elastic compute. These constraints do not disappear with middleware layers or API wrappers. They require architectural resolution. This chapter covers what that resolution looks like in practice - and what happens to AI programmes when it is deferred

What it is actually costing you

63%

of organisations either do not have or are unsure whether they have the right data management practices for AI (Gartner, 2025)

20 to 40%

of the entire technology estate -- that is how much CIOs attribute to technical debt before depreciation, according to McKinsey. 60 % say it is worse than three years ago

40%

of engineering time in legacy-heavy organisations is spent on maintenance and integration work rather than building new capability

The pattern is consistent. An organisation invests seriously in an AI programme. A capable data science team is in place. The model is selected, the tooling is procured, the ambition is genuine. Then the team spends six months trying to access clean, structured, governed data from systems that were never designed to expose it. The AI initiative stalls. The model gets the blame. The real cause is the infrastructure.

Legacy modernisation is not a migration exercise. It is a platform, data, and operating-model problem -- and how you resolve it determines the architecture everything else runs on for the next five years.

The three ways legacy stacks block AI -- specifically

The first is data extraction and integration friction. Legacy systems -- monolithic ERPs, on-premise databases, COBOL-backed mainframes -- were designed to serve the applications that owned them, not to expose data to external consumers. Extraction typically requires bespoke pipelines built against undocumented schemas, with no native API layer and no guaranteed consistency. Those pipelines break on source changes and drift silently when they do not. The result is that data engineering teams spend the majority of their time maintaining access rather than building the foundation AI workloads need.

The second is data semantics and lineage. Even where data can be extracted, it rarely meets the structural requirements AI models depend on. Legacy systems encode business logic inside field values, apply implicit transformations at the point of write, and maintain no separation between raw and processed data. There is no lineage -- no documented record of what has been applied to the data before it reaches a downstream consumer. You cannot build a governed, reliable model on data whose transformation history is unknown or unrecoverable.

The third is fixed compute and deployment constraints. On-premise infrastructure cannot elastically scale to meet AI training or inference workloads. This is not a configuration problem -- it is an architectural one. Cloud is a structural requirement for production AI, not a preference. And migrating workloads to cloud while leaving legacy data models and application dependencies in place resolves the compute constraint only. The data access and semantics problems travel with the application. Lift-and-shift moves the workload. It does not move the architecture.

MERIT

The Four Architectural Patterns That Prevent AI Reaching Production

Not all legacy debt is the same. Across modernisation programmes, the same four patterns appear in different combinations. What they share is a direct impact on AI delivery -- on the availability of data, the reproducibility of pipelines, the deployability of models, and the freshness of features reaching production. Identifying which apply to your stack determines not just what your modernisation programme needs to address, but in what order.

Pattern 01 - The monolithic data store

A single relational database serving as operational store, reporting store and integration hub for every consuming application simultaneously. Everything queries the same database directly. Schema changes require coordinated releases across every system that touches the data. Adding a new consumer -- such as an AI feature pipeline -- adds another direct dependency to an already overloaded system that was never designed to carry this kind of load.

The resolution depends on what the downstream AI workload requires. Where real-time feature freshness is not critical, change data capture into an analytical store is often sufficient -- and a lower-risk starting point than introducing a full event architecture. Where freshness matters and downstream consumers are multiplying, event-driven decoupling becomes the more durable pattern. The sequencing question is whether to start with CDC as a tactical bridge or commit to event architecture from the outset -- and that decision should be driven by the latency requirements of the AI use cases in scope, not by architectural preference. Change data capture streams database changes to an event bus -- Kafka or Kinesis -- and consuming applications read from the event stream rather than the database directly. A purpose-built analytical store handles read workloads without competing with operational traffic, returning the source database to its intended function as the operational store for one application. How data is then served to downstream consumers depends on the use case. Reporting and exploratory workloads may read directly from the analytical store. AI feature pipelines typically require a feature-serving layer with low-latency access and versioning. Domain consumers may be better served through curated data products or domain APIs that abstract the underlying store entirely. The analytical store is the foundation -- not the single consumption point for every system in the estate.

Pattern 02 - Undocumented business logic in the application layer

Twenty years of business rules encoded in application code that no currently active engineer fully understands. Pricing calculations in a COBOL batch job -- where migration requires not just code translation but test harness creation against behaviour that was never formally specified. Transformation logic in a stored procedure that predates the current team -- where the migration risk is dependency isolation, not the rewrite itself. Data quality rules embedded in a Perl scraper that nobody has touched in three years -- where the first task is business-rule externalisation before any modernisation work can begin.

The risk is not that this logic is wrong -- much of it may be entirely correct and business-critical. The risk is that it is opaque, untestable, and unmovable without understanding what it does, what depends on it, and what breaks if it changes.

AI-assisted code analysis is genuinely useful at this stage. Static analysis combined with LLM-based code comprehension can document what legacy code does at a function level, identify logical boundaries for modular extraction, and generate candidate test suites that capture current behaviour before migration begins. This accelerates discovery and decomposition work that would otherwise be entirely manual. It does not replace engineering validation. The output of AI-assisted analysis is a starting point for the engineering team, not a verified specification. How long reverse-engineering takes depends on the complexity and age of the codebase, the availability of any prior documentation, and whether the logic can be tested against known outputs. Weeks is possible in constrained scope. Months is common in practice.

Pattern 03 - The integrated monolith

A single application that does everything: data ingestion, processing, storage, reporting and API serving, all in one codebase with no meaningful internal boundaries. Deploying a change to the reporting module requires testing and releasing the entire system. A failure in the ingestion layer takes down reporting. Scaling the API tier means scaling the entire application, including the components that do not need more resource.

The established modernisation path is strangler fig decomposition: identifying the bounded contexts within the monolith, extracting them one at a time as independently deployable services, and routing traffic through an API gateway that can serve either the legacy monolith or the new service for any given endpoint. The monolith is not replaced in a big bang. It is retired incrementally, with each extracted service validated in production before the next extraction begins.

What makes incremental extraction work safely is the instrumentation around it. Observability must be in place before cut-over -- distributed tracing and error-rate monitoring allow the team to compare extracted service behaviour against the legacy baseline. Contract testing between the gateway and each service catches schema drift and behavioural divergence before it reaches production. Rollback patterns need to be defined per extraction: the gateway provides the routing mechanism, but rollback is only reliable if data state between the legacy system and the new service has been kept consistent during the transition window.

Pattern 04 - The on-premise data warehouse

An on-premise warehouse -- Teradata, Netezza, SQL Server Analysis Services, or an ageing Hadoop cluster -- that was built to serve batch analytics and is now expected to handle real-time AI inference, ML training workloads, and interactive exploration simultaneously. These workloads have fundamentally different compute profiles. Running all three against a fixed, shared estate means every workload is under-served and competing for the same constrained resource.

The architectural requirement is workload isolation through elastic compute separation -- not necessarily a move to public cloud, but a move away from fixed, shared infrastructure where storage and compute are coupled. Object storage with separately scaled compute clusters for batch processing, interactive query, and ML training is achievable in public cloud, private cloud, and regulated hybrid environments. What matters is that each workload class can scale independently without competing for headroom with the others.

For organisations with data residency, sovereignty, or regulatory constraints, the same separation principles apply within private cloud or on-premise Kubernetes environments. The constraint is not location -- it is the coupling of storage to fixed compute. That coupling is what makes AI training runs a negotiation for infrastructure headroom rather than a schedulable workload. Resolving it is what makes a shared data platform viable for AI at scale.

MERIT

What a Genuine Modernisation Strategy Looks Like

A genuine modernisation strategy is not a roadmap. It is a sequence of decisions that determines how the programme will actually execute: in what order, at what risk level, and with what validation criteria at each stage. The decisions that matter most are not which cloud or which target architecture. They are which migration units move first, how legacy and modern systems coexist during the transition, what the validation gate is at each stage, and what constitutes done.

Discovery: Understand what you actually have before you decide where you are going

Modernisation programmes that underestimate legacy complexity almost always do so because discovery was underinvested. A two-week architecture review against available documentation is not discovery. Discovery is the systematic mapping of every system, every data flow, every integration point, and every undocumented dependency in the current estate -- assembled from multiple evidence sources rather than from documentation alone.

The evidence sources that matter in practice are: static codebase analysis to identify component boundaries and coupling patterns; runtime call tracing and application performance monitoring to capture actual traffic flows rather than assumed ones; batch schedule inventories to understand sequencing dependencies between jobs that never appear in architecture diagrams; integration logs to identify which systems are genuinely coupled at the data level; data lineage tooling to trace how data moves between systems and what transformations are applied in transit; release dependency maps to understand which components must be deployed together; and infrastructure inventory to identify shared middleware, shared databases, and network dependencies that constrain extraction sequencing.

No single source is sufficient on its own. Codebase analysis misses runtime behaviour. Runtime monitoring misses batch dependencies. Integration logs miss business workflow dependencies. A credible dependency map requires correlation across all of these sources, with engineering validation at each stage to catch what automated analysis cannot -- implicit contracts between systems, business rules encoded in operational procedures rather than code, and integration points that only activate under specific business conditions.

AI-assisted analysis accelerates the documentation and candidate decomposition phases of this work. Static analysis combined with LLM-based code comprehension can produce structured drafts of component inventories and logical boundaries that would take significantly longer to assemble manually. That output is a starting point, not a verified specification. The dependency map for a modernisation programme is only as reliable as the engineering validation applied to it.
The output of discovery is not a diagram. It is a ranked inventory of migration units -- bounded components that can be moved independently -- with dependency maps, data volume profiles, business criticality ratings, risk assessments, and validation criteria for each one. This inventory is what makes sensible sequencing possible. Without it, sequencing is guesswork dressed up as planning.

Sequencing: **Right order, not fastest order**

The instinct in most modernisation programmes is to start with what looks easiest: low-risk batch jobs, reporting pipelines, internal tools. This is understandable and often wrong. The workloads that need to move first are the ones actively blocking the outcomes the programme exists to deliver -- which in most cases means the data access layer and the core data stores that AI pipelines need to consume cleanly.

A sequencing principle that works in many programmes is to move the data before moving the applications. Establishing a reliable, governed data platform first -- even while legacy applications continue running against on-premise stores -- gives AI teams a clean foundation to build on without waiting for application migration to complete. Change data capture pipelines from legacy sources can run in parallel with operational systems, allowing AI capability to be delivered while the broader migration continues.

This is not a universal rule. Some organisations will need to modernise application interfaces or security and access control layers before governed data extraction is safe or practical. Others will have regulatory constraints that determine sequencing independently of technical preference. The correct sequence is determined by where the constraining bottleneck actually sits -- not by a general principle applied without reference to the specific estate.

> The organisations that get AI into production are not the ones with the most aggressive modernisation timelines. They are the ones that identified their specific bottleneck early, sequenced around it, and gave AI teams something to build on while the rest of the migration ran.

Coexistence: Running old and new without breaking either

The period between starting a modernisation and completing it -- which for large organisations is typically measured in years, not months -- requires deliberate coexistence architecture. Legacy systems and modern systems need to share data reliably during this window, without either system becoming a constraint on what the other can do.

The common pattern is streaming coexistence: change data capture at the legacy source feeds an event stream, which both the legacy adapter and the modern service consume independently. Neither system depends on the other directly. Data consistency is maintained at the event stream level rather than through direct synchronisation between two databases.

Making this reliable in practice requires addressing four concerns that are frequently underspecified at programme outset.

- **Reconciliation:** the event stream and the legacy source will diverge under failure conditions. Reconciliation processes need to be designed and tested before cut-over, not after the first production incident. This means defining how the modern system identifies and resolves discrepancies between its state and the legacy source of record during the coexistence window.
- **Idempotency:** events will be delivered more than once under failure and retry conditions. Every consumer of the event stream must handle duplicate delivery without producing duplicate state changes. This is a design requirement for every service built against the stream, not an edge case to be handled later.
- **Replay:** the ability to reprocess historical events is essential for backfilling the modern system, recovering from processing failures, and onboarding new consumers without requiring a separate data extract. Event retention policies and replay tooling need to be part of the coexistence architecture from the outset, not added when the first recovery scenario arises.
- **Event ordering:** CDC events from a legacy source reflect the ordering of writes in that source. Where the modern system has ordering dependencies -- financial transactions, inventory state changes, workflow progressions -- the event stream architecture must preserve ordering guarantees within the relevant partition boundaries. Assuming ordering without designing for it is a common source of subtle correctness failures that are difficult to diagnose in production.

  When the legacy system is eventually decommissioned, the event stream remains and the modern system continues without disruption -- but only if these four concerns have been resolved during the coexistence period rather than deferred to the decommission phase.

MERIT

AI-Ready Data Architecture: What the Target State Actually Looks Like

The phrase "AI-ready data architecture" appears in almost every modernisation proposal written in the past three years. Most of them mean different things by it. At the architectural level, it means three specific things: the data platform can deliver data at the freshness AI workloads require; the data is governed well enough to support reproducible model training and auditable inference without manual verification; and the compute layer provides elastic workload isolation so that training, inference, and operational traffic do not compete for the same constrained resource. Each has specific architectural implications.

The lakehouse as the foundational layer

The lakehouse pattern -- cloud object storage as the persistence layer, with a table format providing transactional semantics and query reliability on top -- is now the established foundation for AI-ready data platforms. Object storage alone is insufficient: it provides no consistency guarantees, no schema enforcement, and no reliable concurrent write semantics. A table format such as Delta Lake or Apache Iceberg adds exactly those properties without the cost and rigidity of a traditional data warehouse.

Delta Lake integrates most deeply with Spark and Databricks workloads and provides strong write consistency through its transaction log architecture. Apache Iceberg offers the strongest multi-engine compatibility -- Spark, Trino, Flink, Athena, and Hive all read Iceberg tables natively, which matters where multiple teams consume the same data through different compute engines. The choice is not which is better in the abstract. It is determined by workload fit, engine interoperability, governance model, and operating context -- specifically which engines your AI teams are running, how data products are shared across teams, and what governance controls the platform needs to enforce at the storage layer.

Both support time-travel -- the ability to query a table as it existed at a specific point in time. For AI specifically, this capability extends across three operational needs: training-set reconstruction, so that the exact data a model was trained on can be recovered at any point in the past; rollback analysis, so that model performance degradation can be correlated with changes in the underlying data; and audit support, so that regulated use cases can demonstrate what data was used in a given model run and what transformations were applied to it. These are governance and operability requirements that raw data lakes cannot meet.

Data mesh: An organisational model that only works when the foundations are in place

The lakehouse solves the storage and governance problem for a central data team. Data mesh addresses a different constraint: at scale, a centralised data team becomes the bottleneck. Domain teams across the organisation waiting for central engineering to build and maintain their pipelines is a structural speed limit that constrains how fast AI capabilities can be developed.

Data mesh is an architectural and organisational pattern -- not a product -- that distributes data ownership to domain teams while maintaining federated governance standards across all of them. Each domain owns its data products: the pipelines that produce them, the quality standards they meet, the SLAs they commit to. A central platform team provides the infrastructure -- the lakehouse, the event bus, the governance tooling -- but does not own the data flowing through it.

This model only works when three conditions are already in place. Domain teams must be capable of owning data engineering work independently -- organisations without mature domain engineering capability will find distributed ownership produces inconsistency rather than speed. The platform team must be able to build infrastructure that other engineers can use without deep platform expertise -- a self-serve data platform is a product, and it requires product thinking to build and maintain. Governance standards must be enforced automatically by the platform rather than through a manual compliance process -- federated ownership without automated guardrails produces federated governance failures. Where these conditions are not yet met, data mesh is an organisational destination to build toward, not a pattern to adopt immediately.

When all three are in place, the rate at which new AI use cases can be built across the organisation is constrained by ideas, not by engineering bandwidth.

Governance as architecture, not process

The governance failure mode in most modernised platforms mirrors the failure mode in the legacy platforms they replaced: policies exist in documentation and are not enforced in the system. A data catalogue with manually maintained metadata is not governance. Access control rules that require a support ticket are not governance. Retention policies applied through quarterly manual reviews are not governance. None of these prevent a model training job from ingesting PII. None of them stop a schema change from reaching production without validation.

Governance as architecture means the controls are enforced by the platform itself. There are four categories of control that need to be present for a data platform to support governed AI delivery.

- **Lineage.** Every transformation applied to data in transit needs to be recorded automatically, not instrumented manually by individual pipeline authors. OpenLineage provides an open standard for capturing lineage across heterogeneous pipelines. Without automatic lineage, training data provenance is unverifiable and audit responses require manual reconstruction.

- **Access enforcement.** Column-level masking policies in Databricks Unity Catalog, BigQuery column-level access controls, and Snowflake masking policies apply access rules automatically at query time. No application layer needs to enforce them because the compute layer does. This is the only access enforcement model that scales reliably across a large number of consumers and use cases.

- **Policy automation.** PII classification and data sensitivity tagging need to run as part of the ingestion pipeline, not as a manual tagging exercise applied after the fact. Automated classification ensures that sensitive field identification is not dependent on a person remembering to tag a new dataset before it reaches a model training job.

- **Data quality gating.** Quality must be enforced as a pipeline gate, not monitored after data has reached downstream consumers. Bad data should fail the pipeline at the point of ingestion or transformation. By the time a quality issue reaches a model, the cost of remediation is significantly higher than catching it at source.

When these four controls are embedded in the platform, governance stops being a constraint on engineering velocity and starts being a capability. Teams can move faster because they trust the data they are working with. Models can be deployed with confidence because training data provenance is verifiable. Regulated industries can build AI without treating compliance as a blocker.

CHAPTER 5

How to Modernise Without Stopping the Business

The organisations that modernise production systems successfully are not the ones that accept more risk. They are the ones that engineer risk out of the programme through sequencing, coexistence architecture, explicit validation gates, and defined rollback conditions at every stage. The model below is structured around that principle. Each phase has defined entry criteria, operating mechanics, traffic cutover conditions, dual-running requirements, and service-level acceptance criteria. Nothing advances on schedule alone.

The phased migration model that actually works

Phase 01 - Foundation first: data platform before application migration

Establish the cloud data platform before moving any application workloads. The lakehouse, event bus, governance tooling, and data quality framework must be in place and validated before they carry any production dependency.

- **Coexistence operating model during Phase 01.** Change data capture pipelines are instrumented at legacy source systems, streaming changes to the cloud platform in near real time. The legacy application continues running against its existing data store without modification. The CDC pipeline runs in parallel, feeding the cloud lakehouse with governed, quality-validated data. This is the start of dual-running: the legacy system remains the system of record, and the cloud platform runs alongside it as a validated shadow until it is ready to carry production dependency.

  Data quality gates run at the point of ingestion. Records that fail schema validation, referential integrity checks, or completeness thresholds are quarantined and flagged rather than propagated to downstream consumers. Data products exposed from the lakehouse carry explicit SLAs, schema contracts, and freshness guarantees. The governance controls covering lineage, access enforcement, policy automation, and quality gating are operational before any AI workload depends on them.

- **Dual-running validation.** During this phase, outputs from the cloud data platform are continuously reconciled against the legacy source of record. Reconciliation jobs run on a defined schedule, comparing record counts, aggregate values, and key business metrics between the two environments. Discrepancies are alerted, triaged, and resolved before downstream AI workloads are onboarded. Dual-running continues until reconciliation confirms consistent state across all source domains.

- **Acceptance criteria for Phase 01.** CDC pipelines sustaining target latency under peak legacy write volume. Data quality gate pass rates meeting agreed thresholds across all source domains. Reconciliation jobs confirming consistent state between legacy and cloud platforms. At least one AI workload running in production against lakehouse data. Governance controls verified by a platform audit before Phase 02 begins.

Phase 02 - Strangler fig decomposition of monolithic applications

Application extraction begins with service-boundary definition, not with code. Bounded contexts within the monolith are identified through the dependency mapping produced in discovery -- specifically, which logical domains have the clearest data ownership boundaries, the fewest cross-domain dependencies, and the highest value as independently deployable services. Extraction sequencing is determined by dependency isolation, not by perceived ease.

- **Extraction method.** Each extraction follows the same sequence. The service boundary and its data ownership are defined before any code is written. Consumer-driven contract tests are written against the existing monolith behaviour before extraction begins, capturing the interface contract the extracted service must honour. The extracted service is deployed behind the API gateway before it receives any production traffic. Observability -- distributed tracing, structured logging, latency and error-rate monitoring -- is instrumented and baselined against the legacy monolith before cutover begins.
- **Traffic cutover.** Cutover is a staged process, not a single event. The API gateway is configured to route a defined percentage of traffic -- typically one to five percent initially -- to the extracted service while the remainder continues to the legacy monolith endpoint. Traffic percentage is increased incrementally against defined thresholds: each increment requires the extracted service to sustain its service-level acceptance criteria over a defined validation window before the next increment proceeds. Full cutover to the extracted service only occurs after the service has been validated at one hundred percent of production traffic volume. The legacy monolith endpoint is not decommissioned until the full validation window has elapsed and all acceptance criteria have been met.
- **Rollback.** Rollback capability is maintained at every increment of the cutover sequence. At any point before legacy endpoint decommission, traffic can be routed back to the monolith via the API gateway without application changes. Rollback is triggered automatically if error rates or latency thresholds breach defined limits during the cutover window. The conditions that trigger automatic rollback are defined and tested before the first increment of production traffic is routed to the extracted service.

- **Validation criteria.** Traffic percentage is one signal. The full validation set covers: data correctness -- the extracted service must produce outputs behaviourally equivalent to the monolith for the same inputs, verified through automated comparison testing; downstream dependency validation -- every system consuming the endpoint must be confirmed to behave correctly against the new service; and service-level acceptance criteria -- latency, error rate, and throughput targets sustained under full production load over the agreed validation window.
- **Dual-running during Phase 02.** Where the extracted service maintains its own data store, dual-writing to both the legacy database and the new service data store runs during the validation window. Reconciliation jobs compare state between the two stores on a scheduled basis, with discrepancy alerting and documented resolution procedures. Dual-running ends only after the legacy endpoint is decommissioned and the legacy data store is no longer the system of record for that domain.

Phase 03 - Data model modernisation

Once application services are decoupled from the legacy database, the data model can be modernised without risk to operational traffic. This phase has four components that must be sequenced carefully.

- **Schema harmonisation.** Legacy schemas were designed around application requirements, not analytical or AI query patterns. Harmonisation maps legacy field semantics to a target schema, resolving naming inconsistencies, implicit type conversions, and business logic embedded in field values. The harmonised schema is validated against known query patterns from the AI and analytics workloads already running on Phase 01 data products before historical migration begins.
- **Historical backfill.** Bulk migration jobs run against the harmonised schema in bounded batches with row-count reconciliation, checksum validation, and business-rule verification at each batch boundary. Backfill jobs are designed to be resumable -- failures restart from the last validated checkpoint rather than from the beginning.
- **Master data impacts.** Entities appearing across multiple legacy source systems -- customers, products, counterparties, locations -- frequently carry inconsistent identifiers, duplicate records, and conflicting attribute values. Master data reconciliation must be resolved before those entities are migrated, not discovered after consuming services start producing incorrect join results against the modern store.

- **Dual-running and reconciliation.** The legacy database remains the system of record until every consuming service has been validated against the modern store. Reconciliation jobs compare state between the two stores on a scheduled basis throughout the migration window, with discrepancy alerting and documented resolution procedures. The legacy database is decommissioned only after reconciliation confirms consistent state across all domains and all consuming services have passed their acceptance criteria against the modern store over the agreed validation window.

Phase 04 - AI capability build at scale

With governed, cloud-native data and independently scalable services in place, this phase builds out the AI capabilities the earlier phases were designed to enable. The point is not simply that AI can now be built. It is that every component the platform depends on -- feature pipelines, training data, inference services, and governance controls -- now operates without the brittle workarounds that characterised AI delivery against legacy infrastructure.

Feature engineering pipelines consume from lakehouse data products with guaranteed freshness SLAs, schema contracts, and lineage tracing every transformation back to the legacy source. ML training runs on elastic compute that scales to workload without competing with operational traffic. Real-time inference is served via cloud-native endpoints with latency guarantees that fixed on-premise infrastructure could not support. Training datasets are reproducible via time-travel, with full audit trails for regulated use cases.

Governance controls enforce data access, PII handling, and quality standards automatically at the platform layer -- not through manual process applied inconsistently across teams. New use cases can be onboarded against existing governed data products without rebuilding access controls or re-validating data quality from scratch. This is what allows AI delivery to scale across the organisation without compliance becoming the bottleneck.

The modernisation programme delivers what it was scoped to deliver: a platform that supports AI in production, not infrastructure that is simply newer than what it replaced.

Three Production Modernisations

The following are drawn from Merit's modernisation work. They were selected because each represents a distinct class of legacy problem and a different modernisation approach: platform unification across fragmented operational systems, cloud-native transformation to enable ML and analytics at scale, and AI-assisted code modernisation of a large legacy codebase. Together they illustrate the range of decisions, trade-offs, and sequencing logic that production modernisation programmes require -- described at a level of detail that gives an honest picture of what the work involved, not a headline summary of what it achieved.

AgTech: From siloed legacy infrastructure to a unified, self-serve data platform

A global agri-intelligence company -- providing data, analytics, supply chain automation and agronomy services internationally -- was running into a structural data problem that growth had made impossible to ignore. Data silos across business units meant the same data was being maintained in multiple places, with different quality standards and no clear source of truth. Rapid acquisition activity was compounding the problem. The infrastructure could not handle semi-structured and unstructured data at the volumes the business was now generating, and the escalating cost of maintaining legacy systems was becoming a line item that the board could see. Their ambition was to become a self-serve analytics organisation. Their current infrastructure made that aspiration structurally unreachable.

- **What we delivered:** The modernisation programme addressed four specific problems. Ingestion standardisation: a common ingestion framework was established across all source systems and acquired entities, with schema validation, quality gating, and lineage capture running automatically at the point of entry rather than applied inconsistently downstream. Governance controls: column-level access policies, PII classification, and data quality thresholds were embedded in the platform layer, enforced automatically rather than maintained through manual process. Self-serve enablement: a data product layer was built on top of the governed lakehouse, exposing certified, SLA-backed datasets that domain teams could access and build on without central engineering involvement. AI-readiness: the platform was structured from the outset to support ML workloads -- clean, versioned, lineage-tracked data products with the freshness and reproducibility guarantees that feature pipelines and model training require, not just the reporting and dashboarding the previous infrastructure had been built for.

- **The outcome:** 65% reduction in data quality issues through unified platform governance. 40% increase in report usage across the organisation -- a direct measure of whether self-serve actually worked, not just whether it was built. 60% reduction in storage costs through efficient cloud-native data management replacing duplicated on-premise stores. 70% faster data onboarding for new sources, meaning acquisitions and new data partnerships could be absorbed without infrastructure delays that had previously slowed the business.

Automotive: From structurally impossible to production-ready ML -- a cloud and data transformation

A global automotive business intelligence and analytics provider -- specialising in vehicle analytics and market intelligence -- had reached the limit of what their current infrastructure could do. There was no centralised data platform: reporting and analytics applications pulled from siloed sources that disagreed with each other, which meant reporting errors and decision-making gaps that showed up in the quality of the intelligence the business was selling. The legacy technology stack made AI and ML adoption structurally impossible -- not difficult, impossible. There was no path to deploying ML models in production on the infrastructure that existed. The business knew it needed to change. It needed a partner that could deliver the technical transformation and manage the transition without disrupting a business that ran on the intelligence it was producing.

- **What we delivered:** Merit delivered an end-to-end data and ML transformation. The primary outcome was ML deployment capability: taking an organisation where ML in production was structurally unreachable to one where new use cases could be adopted 70% faster than before. The centralised data platform, cloud enablement of all data assets, and DevSecOps enablement -- CI/CD pipelines, infrastructure as code, automated deployment -- were the architectural enablers that made that possible. ML-driven improvements to data mapping accuracy directly improved the quality of the intelligence product, which was the commercial outcome the programme was ultimately accountable for. Change management and training were delivered alongside the technical work to make sure the organisation could sustain and build on what had been put in place.
- **The outcome:** 60% time and cost efficiency improvement, delivered partly through AI-assisted code conversion tools applied to the migration itself. 32% faster assessment and design timelines through consulting accelerators built from previous engagement experience. 70% reduction in new use case adoption time -- the infrastructure that had blocked new ML deployments was replaced by infrastructure that accelerated them. 40% reduction in IT infrastructure costs through cloud migration replacing on-premise systems the business had been maintaining at significant cost.

Construction: AI-led code modernisation from legacy PERL to production Python

The UK's leading construction intelligence provider ran a legacy PERL scraper application at the core of their data acquisition operation. It had been doing the job for years, but the constraints had become impossible to work around. Growing data volumes and the increasing variety of document formats online were pushing the application beyond its scalability limits. Dynamic website changes required constant manual maintenance from developers who had better things to do. The absence of asynchronous support created processing bottlenecks that slowed the entire data pipeline. The codebase could not support the GenAI-driven automation use cases the business wanted to build -- and increasingly needed to build to stay competitive. The cybersecurity and compliance exposure from an unmaintainable legacy codebase was growing every quarter. The question was not whether to modernise. It was how to do it without breaking the data acquisition operation the business depended on..

- **What we delivered:** Merit carried out a complete overhaul of the core application, structured around four engineering workstreams. Modular redesign: the monolithic PERL architecture was decomposed into independently deployable Python modules with defined boundaries, replacing a codebase where components were tightly coupled and changes carried unpredictable blast radius. Conversion controls: feature-by-feature code conversion used AI-assisted migration tools with prompt engineering frameworks specifically designed to produce idiomatic Python rather than transliterated PERL -- enforcing modern Python standards, asynchronous processing patterns, and clean separation of concerns at each conversion stage. Test automation: automated test suites were written against existing PERL behaviour before conversion began, providing behavioural equivalence validation at each feature boundary and preventing regression from reaching production. CI/CD and DevSecOps: deployment through Merit's DevSecOps frameworks replaced a manual release process that had been a consistent bottleneck, with infrastructure as code and automated pipeline stages covering build, test, security scanning, and deployment. The modernised Python target state was designed from the outset for extensibility -- specifically to support the asynchronous processing and GenAI-driven automation use cases the legacy codebase had made structurally unavailable.
- **The Outcome:** 30% reduction in operational costs through modern, efficient Python code replacing a PERL codebase that required high maintenance overhead. 40% improvement in developer productivity -- faster prototyping, debugging and scaling on a codebase the team could actually work in confidently. 35% faster time-to-market for new product releases through CI/CD deployment replacing a manual release process that had been a bottleneck. 30% improvement in system reliability, reducing the downtime that the legacy application had been causing with increasing frequency.

CHAPTER 7

Is Your Stack Ready to Modernise? A Diagnostic

These questions are designed to surface the specific constraints in your current architecture -- the ones that will determine how a modernisation programme needs to be sequenced and what it will actually require to execute. Enterprise stacks rarely fail these questions completely or pass them cleanly. The value is in identifying where partial implementations, legacy exceptions, and architectural debt are concentrated -- and what that means for sequencing. Work through them against your production stack, not your roadmap.

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.

Can your AI and ML teams access production data without depending on the application team that owns the source system?

Fully in place means a governed data platform exposes certified data products that AI teams can consume without routing requests through application owners. Partially in place typically means some domains have self-serve access while others still require manual exports, tickets, or direct database queries that compete with operational traffic. Not in place means data access is consistently mediated by application teams regardless of domain. The partial state is common and manageable -- but the domains still requiring manual access are the ones that will constrain your AI programme first.

Do you have change data capture instrumented on your core operational databases, or does all data movement rely on scheduled batch extracts?

Fully in place means CDC is running on core operational sources with defined latency SLAs feeding downstream consumers. Partially in place means some sources have CDC while others -- typically older or more sensitive systems -- still rely on batch. Not in place means all data movement is batch-dependent. The partial state is a sequencing input: the sources without CDC are the ones that will determine feature freshness limits for any AI workload that depends on them

Is your transformation logic tested, versioned and deployable through a CI/CD pipeline -- or does it live in stored procedures, ad-hoc scripts or application code that predates the current team?

Fully in place means all transformation logic is version-controlled, tested, and deployed through automated pipelines with validation gates. Partially in place means modern pipelines coexist with legacy stored procedures or scripts that have not yet been migrated -- a state that is nearly universal in large estates. Not in place means transformation logic is predominantly undocumented and undeployable safely. The partial state requires an inventory: which legacy transformations carry the most business-critical logic, and which present the highest risk of silent drift.

Do you have a cloud-native lakehouse serving your analytical and AI workloads, or are those workloads hitting an on-premise warehouse or an unstructured raw data lake?

Fully in place means a governed lakehouse with transactional semantics, elastic compute separation, and workload isolation is serving analytical and AI consumers. Partially in place covers a range of common states: cloud migration underway but not complete, a lakehouse in place but without governance controls, or elastic compute available for some workload classes but not others. Not in place means AI and analytical workloads are competing for fixed on-premise infrastructure. Each partial state has different sequencing implications -- the constraint is not always the storage layer

Is access control and PII governance enforced programmatically in your data platform, or managed through application-level controls and manual compliance processes?

Fully in place means column-level access policies, PII classification, and data masking are enforced automatically by the platform at query time. Partially in place means platform-level controls exist for some data domains or sensitivity tiers while others rely on application-level enforcement or manual process. Not in place means governance is entirely procedural. The partial state is the most common enterprise position -- and the gaps are usually concentrated in the domains where legacy data stores have not yet been brought under platform governance.

Can you independently deploy, scale and roll back individual components of your data platform without affecting other components?

Fully in place means ingestion, transformation, and serving layers are independently deployable with no coordinated release dependency between them. Partially in place means some components have independent deployability while others remain coupled -- a common state during active modernisation. Not in place means platform changes require coordinated deployment across multiple layers. Where coupling exists, it is a sequencing constraint: the coupled components need to be decoupled before the modernisation of either can proceed safely

Do automated quality gates in your pipelines fail the job when data quality standards are not met -- or is quality monitored after data has already reached downstream consumers?

Fully in place means quality is enforced as a pipeline gate at ingestion, transformation, and serving, with violations failing 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 post-hoc, after data has reached consumers. The partial state requires mapping which pipelines feeding AI workloads are ungated, as those are the ones carrying silent quality risk.

Four or more uncertain answers means your infrastructure has debt that is already constraining your AI programme, whether or not that constraint is currently visible in your metrics. The gap is fixable. But it needs to be fixed at the architecture level, not worked around at the team level.

Reading the results

A pattern of fully in place responses indicates an estate that can support a modernisation programme without resolving foundational blockers first. A pattern of partially in place responses -- which is the most common enterprise position -- indicates that sequencing decisions need to be driven by where the gaps are concentrated relative to the AI use cases in scope, not by a uniform modernisation sweep. A pattern of not in place responses across multiple dimensions indicates that foundational data platform work is a prerequisite before application migration can proceed without compounding existing debt.

The diagnostic is not a pass/fail threshold. It is a sequencing input. The gaps it surfaces are the constraints your programme needs to address -- in the order that most directly unblocks the AI outcomes you are trying to deliver.

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