International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

# Modernizing Legacy ETL with Cloud-Native Lakehouse Architecture: Migration Strategies from Traditional Integration Platforms

**Anil Kumar Kolla**

Department of IT, PiSquare, *USA*
*akkolla1988@gmail.com*

**Abstract:**
Enterprises operating legacy extract-transform-load platforms face a recurring dilemma: the platforms are business-critical, poorly documented, expensive to license and scale, and increasingly incapable of supporting analytic and machine learning workloads, yet wholesale replacement carries unacceptable risk. This paper presents a migration methodology for moving from traditional GUI-based integration platforms to a cloud-native lakehouse architecture. Rather than treating migration as a translation exercise, we frame it as incremental strangulation of a legacy system under continuous parallel verification. The methodology comprises six phases — inventory and dependency reconstruction, value-and-risk stratification, target pattern definition, incremental migration with dual-run reconciliation, cutover, and decommissioning — with explicit entry and exit criteria for each. We contribute a job classification scheme that predicts migration difficulty from observable properties of legacy artefacts, a reconciliation protocol specifying how equivalence between legacy and target outputs is established and what tolerances are defensible, and a technical debt accounting model distinguishing debt eliminated, debt migrated, and debt newly created. We further catalogue the operational risks that recur in such migrations — semantic drift in undocumented transformations, hidden scheduling dependencies, silent type coercion differences, and the erosion of institutional knowledge as legacy maintainers depart — and propose mitigations. An evaluation protocol is specified covering delivery, performance, cost, quality, and organisational measures.

**Index Terms:** ETL modernization, data migration, lakehouse architecture, cloud migration, technical debt, strangler pattern, data engineering, legacy systems.

## I. Introduction

Legacy integration platforms accumulate a specific and unpleasant form of debt. A job authored years ago in a graphical designer encodes business rules that were never documented elsewhere; the author has left; the job runs nightly and is correct as far as anyone knows. Multiply this by several hundred jobs, add undocumented scheduling dependencies and a licensing model that penalises growth, and the result is a system that is simultaneously indispensable and unmaintainable.

The pressure to modernise is well founded. Traditional ETL platforms typically execute transformations on dedicated servers with vertical scaling limits, produce artefacts that are difficult to version and review, integrate poorly with modern testing and CI practice, and have no natural path to machine learning workloads. Cloud lakehouse platforms [1], [2] address these constraints with elastic distributed compute

[3], code-based transformation amenable to version control and automated testing, and native support for both analytic and ML workloads.

The pressure to *not* modernise is equally well founded. These pipelines feed regulatory reporting and operational decisions. A migration that silently changes a number is a serious incident, and the changes most likely to occur silently — type coercion differences, null handling, rounding, timezone interpretation, sort stability affecting deduplication outcomes — are precisely those that no one thinks to check.

This paper's position is that the risk is managed not by better upfront analysis but by structural choices: migrating incrementally, running old and new in parallel, and reconciling outputs empirically before cutover. Analysis of legacy jobs is necessary but never sufficient, because the specification is the running system, not the documentation.

Contributions:

1) **A six-phase migration methodology** with explicit entry and exit criteria.
2) **A job classification scheme** predicting migration difficulty from observable legacy artefact properties.
3) **A reconciliation protocol** defining equivalence classes and defensible tolerances.
4) **A technical debt accounting model** separating debt eliminated, migrated, and created.
5) **A risk catalogue with mitigations**, drawn from failure modes that recur across such migrations.

**Contribution type and scope.** This is a methodology contribution grounded in migration practice. It specifies phases, entry and exit criteria, a difficulty classification, a reconciliation protocol, and a debt accounting model. It does not report a case study, and the methodology has not been compared experimentally against alternative migration approaches. Section VIII gives the pre-specified reporting template so that a case study can be reported in a form comparable across organisations; a worked case study is the natural next contribution and is not claimed here.

## II. Background and Related Work

### *A. ETL and Data Warehousing*

Vassiliadis [4] surveys ETL technology, conceptual modelling, and optimisation, establishing the vocabulary of extraction, transformation, and loading as a distinct engineering discipline. Kimball and Ross [5] define the dimensional modelling practice that most legacy warehouse ETL implements — surrogate key management, slowly changing dimensions, fact table grain — and understanding which of these patterns a legacy job implements is a prerequisite for migrating it correctly.

### *B. Lakehouse Architecture*

The lakehouse [1] combines open-format object storage with transactional table semantics [2], enabling warehouse-style guarantees without warehouse-style storage economics. For migration purposes, three properties are decisive: schema enforcement, which catches the type-coercion divergences discussed in Section VI; time travel, which permits reproducible comparison between legacy and target output for a given date; and atomic commits, which allow a failed load to leave no partial state. Distributed execution [3] removes the vertical scaling ceiling that motivates many migrations in the first place.

### *C. Technical Debt*

The technical debt metaphor originates with Cunningham [6] and was developed into a research agenda by Kruchten *et al.* [7]. Sculley *et al.* [8] extend the analysis to data and ML systems, identifying data

dependencies as the dominant and least visible debt category. The distinction that matters for migration is between debt that is *eliminated* by moving to a new platform, debt that is merely *relocated*, and debt that is *created* by the migration itself — a distinction rarely made in migration business cases, which typically claim only the first.

### *D. Migration Patterns*

The strangler fig pattern [9] — incrementally routing functionality to a new system while the legacy system continues to operate, until the legacy system can be removed — is the dominant approach for high-risk legacy replacement, and is developed in detail for system decomposition by Newman [10]. Systematic reviews of cloud migration research [11] document decision frameworks and recurring challenges. Continuous delivery practice [12] supplies the automated testing and deployment discipline that makes incremental migration tractable.

### *E. Gap*

The literature addresses application migration and cloud migration generally, and ETL as a design discipline, but does not provide a methodology specific to data pipeline migration where correctness must be demonstrated numerically rather than functionally. Data pipelines differ from applications in that "working" is not observable from behaviour — a pipeline that produces wrong numbers runs exactly as successfully as one that produces right numbers. This paper addresses that distinction.

## III. MIGRATION METHODOLOGY

*Phase 1 — Inventory and Dependency Reconstruction*

*Entry:* decision to migrate. *Exit:* complete inventory with reconstructed dependency graph and execution profile.

Legacy artefacts are parsed programmatically rather than reviewed manually. For most integration platforms, job definitions are stored in a structured, machine-readable form from which sources, targets, transformation components, and parameters can be extracted at scale. Manual review does not scale to several hundred jobs and produces inconsistent results.

The dependency graph must be reconstructed from three sources, because none is complete alone: declared scheduler dependencies; inferred data dependencies where one job's target is another's source; and *temporal* dependencies, where jobs are coupled only by the fact that one is scheduled after another and someone once observed that this was necessary. The third category is the most dangerous, is invisible in job definitions, and is best detected by analysing execution logs for ordering constraints that are never violated.

Execution profiling — runtime, data volume, failure frequency, and last-modified date — is captured to support stratification. Jobs not executed within the observation window are candidates for retirement rather than migration; the proportion of a legacy estate that is dead code is routinely underestimated.

*Phase 2 — Value and Risk Stratification*

*Entry:* complete inventory. *Exit:* prioritised migration backlog with sequencing rationale.

Jobs are scored on migration value (cost, runtime, failure rate, blocking of downstream modernisation) and migration risk (downstream consumer criticality, transformation complexity, documentation quality, availability of the original author). Sequencing places low-risk, high-value jobs first to build team

capability and demonstrate benefit, defers high-risk jobs until reconciliation tooling is mature, and retires zero-value jobs without migrating them.

*Phase 3 — Target Pattern Definition*

*Entry:* stratified backlog. *Exit:* documented target patterns with reference implementations.

Migration is not translation. A legacy job implementing a slowly changing dimension load through a sequence of lookup and conditional components should be reimplemented as an idiomatic merge operation on a transactional table, not as a component-by-component transliteration. Transliteration reproduces the legacy design's constraints in an environment that does not have them, and produces code that is harder to maintain than either the original or an idiomatic rewrite.

A small set of target patterns — full refresh, incremental append, merge/upsert, type-2 dimension, deduplication, late-arriving data handling, reference data lookup — covers the majority of jobs in most estates. Each pattern is implemented once as a parameterised reference implementation with tests, and jobs are mapped to patterns during migration.

*Phase 4 — Incremental Migration with Dual-Run Reconciliation*

*Entry:* target patterns available; reconciliation harness operational. *Exit:* migrated job passing reconciliation for the required number of consecutive cycles.

Each migrated job runs in parallel with its legacy counterpart, writing to a shadow target. Outputs are reconciled automatically per Section V. Legacy remains authoritative throughout. This phase is where the methodology's risk reduction actually resides; the preceding phases are preparation for it.

*Phase 5 — Cutover*

*Entry:* sustained reconciliation success. *Exit:* target authoritative; legacy job disabled but retained.

Cutover redirects consumers to the target output and disables the legacy job without deleting it. Rollback is a scheduler change, and this must remain true for a defined period after cutover.

*Phase 6 — Decommissioning*

*Entry:* stability period elapsed with no rollback. *Exit:* legacy artefacts archived; licences released.

Decommissioning is frequently deferred indefinitely, which forfeits the cost benefit that justified the migration. It should be scheduled as work, with the archived artefacts retained for reference.

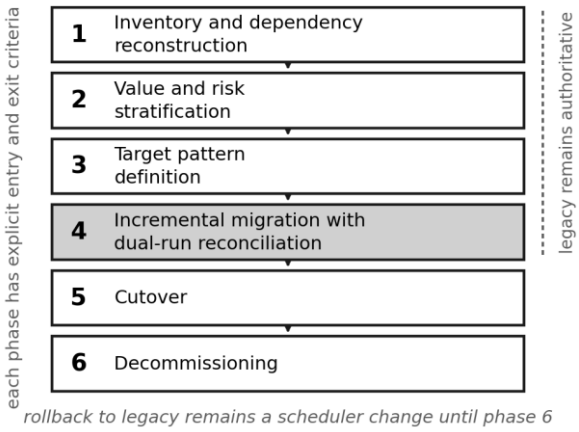



Fig. 1. Six-phase migration methodology. The legacy system remains authoritative through phase four, so rollback stays inexpensive until reconciliation has been sustained.

## IV. Job Classification for Difficulty Estimation

We propose classifying legacy jobs by observable properties, since estimation from job count alone is unreliable — a small number of complex jobs typically dominates total effort.

| Class | Observable properties | Expected difficulty |
|---|---|---|
| A — Mechanical | Simple source-to-target movement, minimal transformation, single source | Low; largely automatable |
| B — Patterned | Implements a recognised warehouse pattern with standard components | Low–moderate; map to reference pattern |
| C — Logic-heavy | Many conditional and derivation components; business rules embedded in expressions | Moderate–high; requires rule extraction and validation |
| D — Opaque | Contains custom code, stored procedure calls, or external scripts | High; semantics not statically determinable |
| E — Entangled | Multiple undocumented downstream consumers, temporal dependencies, or shared mutable staging tables | Highest; risk dominated by dependencies, not logic |

Class D and E jobs should not be scheduled early. Class E in particular is frequently misclassified as simple, because its complexity is external to the job definition; dependency reconstruction in Phase 1 is what surfaces it.

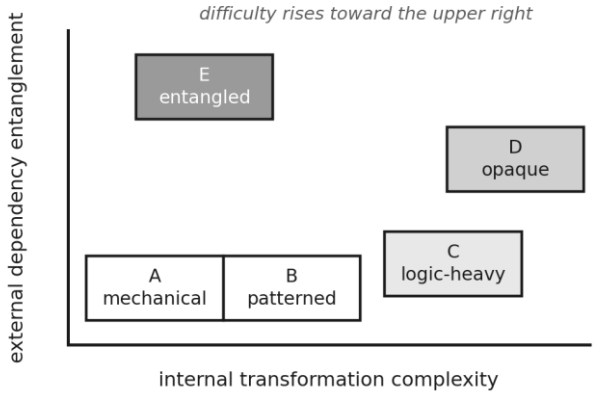



Fig. 2. Job classification. Class E difficulty arises from dependencies external to the job definition and is therefore routinely underestimated.

## V. Reconciliation Protocol

Reconciliation establishes that the target output is equivalent to the legacy output. We define three equivalence levels.

**L1 — Structural.** Row counts, column sets, and data types match. Necessary, weak.

**L2 — Aggregate.** Sums, counts, minima, maxima, and distinct counts match for all numeric and key columns, within stated tolerance. Detects most gross errors while being cheap to compute on large volumes.

**L3 — Record-level.** Full row-by-row comparison on the natural key, reporting differing rows with per-column deltas. Required for financial, regulatory, and clinical outputs; expensive but non-negotiable for those classes.

**Tolerances.** Exact match is required for counts, keys, and categorical values. For floating-point aggregates, a relative tolerance may be defensible, but must be justified by the arithmetic — a difference caused by a different summation order in a distributed engine is acceptable; a difference caused by different null handling is a defect wearing the same clothes. Every tolerance applied must be documented with its cause. A reconciliation regime that applies a blanket tolerance to make failures disappear provides no assurance.

**Known-divergence register.** Some divergences are intentional — a legacy defect being corrected, or a definition being deliberately changed. These are registered explicitly, with approval from the output's business owner, and excluded from failure counts. Undocumented divergence is never accepted as intentional after the fact.

**Sustained passing.** A single passing reconciliation is insufficient. Cutover requires passing across a defined number of consecutive cycles including at least one month-end or period-close cycle, since period-boundary logic is a common source of divergence that daily testing never exercises.

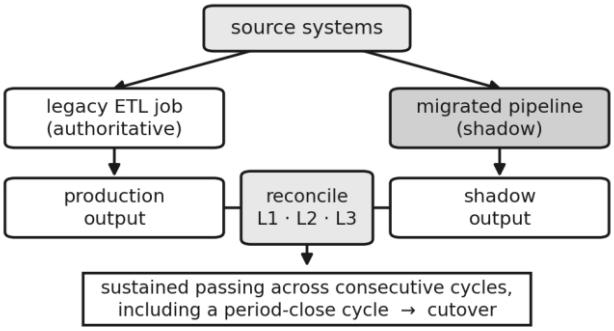



Fig. 3. Dual-run reconciliation. Legacy and migrated pipelines consume the same sources, and equivalence is established empirically before cutover.

## VI. Recurring Risks and Mitigations

**Semantic drift in undocumented transformations.** A legacy expression may implement a rule nobody remembers. Mitigation: reconstruct rules from the legacy artefact, validate the reconstruction against production output, and *record the rule in the target implementation as documented code* — this is the migration's principal durable benefit and is lost if the rule is transliterated without being understood.

**Hidden temporal dependencies.** As described in Phase 1. Mitigation: execution-log analysis for invariant ordering; conservative sequencing where uncertainty remains.

**Silent type and null divergence.** Legacy platforms and distributed engines differ in implicit type coercion, null propagation in aggregates and comparisons, string trimming, and empty-string versus null treatment. These produce small, plausible, invisible differences. Mitigation: L3 reconciliation with explicit null-handling assertions; schema enforcement on target tables.

**Timezone and calendar semantics.** Date boundaries computed in different timezones shift records between periods. Mitigation: explicit timezone declaration in target code; period-boundary reconciliation cycles.

**Sort instability affecting deduplication.** Where legacy logic deduplicates by retaining "the first" record under an unstable sort, the target may retain a different record — a genuinely non-deterministic legacy behaviour that cannot be reproduced, only replaced by a deterministic rule. Mitigation: identify these cases explicitly and agree a deterministic tie-break rule with the business owner as a registered intentional divergence.

**Knowledge erosion.** Legacy maintainers leave, often accelerated by the announcement of a migration. Mitigation: front-load knowledge capture for Class C–E jobs; retain legacy expertise contractually through Phase 4.

**Scope creep into redesign.** The temptation to fix data models during migration is strong and destroys the reconciliation basis, since output is no longer expected to match. Mitigation: strict separation — migrate first with equivalence preserved, redesign afterwards as separate work with its own justification.

**Cost surprise.** Elastic compute removes capacity constraints and, with them, the natural cost ceiling. Mitigation: cost monitoring and budget alerting from the first migrated job, not after the estate has moved.

## VII. Technical Debt Accounting

Migration business cases typically claim debt reduction without decomposition. We propose reporting three categories:

**Debt eliminated.** Licence costs removed; capacity constraints lifted; undocumented rules now expressed as reviewed, versioned code; manual deployment replaced by automated pipelines; absence of tests replaced by test coverage.
**Debt migrated.** Data model deficiencies carried forward unchanged; business rules faithfully reproduced including rules that are wrong; unnecessary jobs migrated rather than retired; naming and structural conventions inherited from the legacy design.
**Debt created.** New platform-specific knowledge concentrated in few individuals; a dual-run period with two systems to maintain; new operational cost variability; and — where transliteration occurred — code that is idiomatic to neither platform.
Honest accounting across all three is what makes post-migration retrospectives useful and is the basis for the organisational evaluation measures below.

## VIII. Evaluation Protocol

Implementers should report, measured on their own estate:

- **Delivery:** jobs migrated per period by class; effort per job by class; schedule variance. Reporting by class is essential, since aggregate velocity is dominated by the class mix in each period.
- **Correctness:** reconciliation pass rate at each equivalence level; count and category of registered intentional divergences; post-cutover defects attributable to migration, tracked for a defined period.
- **Performance:** end-to-end pipeline runtime before and after; critical-path completion time against business deadlines.
- **Cost:** total cost of ownership decomposed into licence, infrastructure, elastic compute, and engineering labour, measured across the dual-run period rather than at a point, since dual-run temporarily increases cost.
- **Quality and maintainability:** test coverage; deployment frequency and lead time [12]; change failure rate.
- **Organisational:** proportion of engineers able to modify migrated pipelines independently; documentation coverage of business rules.

*Pre-Specified Reporting Template*

A case study reported in the form of Table II is comparable across organisations; narrative accounts generally are not. The table is unpopulated here because this paper does not report a case study.

| Measure | Before migration | After migration |
|---|---|---|
| Jobs in scope (retired / migrated) | — | — |
| Licence and infrastructure cost | — | — |
| Critical-path completion time | — | — |
| Pipeline failures per month | — | — |

| Measure | Before migration | After migration |
|---|---|---|
| L3 reconciliation pass rate | — | — |
| Registered intentional divergences | — | — |
| Post-cutover defects attributable to migration | — | — |
| Rollback incidents | — | — |
| Engineering effort per job, by class | — | — |
| Deployment frequency and lead time | — | — |

TABLE II. PRE-SPECIFIED REPORTING TEMPLATE FOR A MIGRATION CASE STUDY. BASELINE MEASUREMENT BEFORE PHASE 1 IS A PREREQUISITE.

Baseline measurement before Phase 1 is a prerequisite. Migrations frequently cannot demonstrate benefit because the pre-migration state was never measured.

## IX. CONCLUSION

This paper presented a methodology for migrating legacy ETL estates to cloud-native lakehouse architectures, structured around incremental strangulation with continuous empirical reconciliation. Its central argument is that in data pipeline migration, correctness cannot be established by inspection — the legacy system is its own specification — and therefore the methodology's risk control must be structural: parallel operation, automated output comparison at a defensible equivalence level, sustained passing across period boundaries, and reversible cutover. Secondary contributions include a difficulty classification predicting effort from observable artefact properties, and a debt accounting model that resists the common overstatement of migration benefit.

Future work should address automated extraction and formal representation of business rules from legacy artefacts, which remains the largest manual effort component; statistical methods for reconciling non-deterministic legacy behaviour; and longitudinal study of whether post-migration maintainability gains persist beyond the initial period.

## REFERENCES:

[1] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *Proc. Conf. Innovative Data Systems Research (CIDR)*, 2021.

[2] M. Armbrust *et al.*, "Delta Lake: High-performance ACID table storage over cloud object stores," *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020.

[3] M. Zaharia *et al.*, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
[4] P. Vassiliadis, "A survey of extract–transform–load technology," *International Journal of Data Warehousing and Mining*, vol. 5, no. 3, pp. 1–27, 2009.
[5] R. Kimball and M. Ross, *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd ed. Indianapolis, IN, USA: Wiley, 2013.
[6] W. Cunningham, "The WyCash portfolio management system," in *Addendum to Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1992, pp. 29–30.
[7] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Software*, vol. 29, no. 6, pp. 18–21, 2012.
[8] D. Sculley *et al.*, "Hidden technical debt in machine learning systems," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 2503–2511.
[9] M. Fowler, "StranglerFigApplication," 2004. [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html
[10] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. Sebastopol, CA, USA: O'Reilly Media, 2019.
[11] P. Jamshidi, A. Ahmad, and C. Pahl, "Cloud migration research: A systematic review," *IEEE Transactions on Cloud Computing*, vol. 1, no. 2, pp. 142–157, 2013.
[12] L. Chen, "Continuous delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.
[13] E. Breck, N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data validation for machine learning," in *Proc. Conf. Systems and Machine Learning (SysML/MLSys)*, 2019.