International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(23s) 2026 pp. 2069-2079
https://doi.org/10.70917/ijcisim-2026-5863
Article

Cloud-to-Cloud Integration Migration Accelerator: A Schema-Agnostic Translation Engine for Workload Migration Across Heterogeneous iPaaS Platforms

Gowrunaidu Seelanki

Independent Researcher, USA

Abstract: Enterprises that move an integration workload from one iPaaS platform to another routinely discover that the effort resembles a rewrite more than a configuration change. Workflow structure, connector behavior, transformation syntax, and error-handling semantics rarely line up between platforms such as Dell Boomi, MuleSoft, and AWS Step Functions, and a migration team is left reconstructing flows by hand, one at a time. This paper proposes a schema-agnostic migration accelerator built around an Integration Intermediate Representation (IR) that records a flow's structure, data contracts, and failure behavior in a platform-neutral graph before target-platform artifacts are generated from it. The approach deliberately separates predictable structural and transformation mappings from constructs that require engineering review, custom connectors, embedded code, and platform-specific runtime behavior and produces a remediation report for the latter instead of forcing a silent, possibly incorrect conversion. We evaluate the approach using a reproducible, explicitly simulated benchmark on a representative synthetic corpus of 419 flow components across seven categories, achieving an aggregate automation rate of 68.5% and an estimated effort reduction of 69.9% compared to fully manual reconstruction, with automation focused on routing, simple transformation, and subprocess-reference constructs, and concentrated failure in custom connectors and parallel branches. We report these findings as an illustrative proof of methodology, not a production result, and outline the real-world pilot required before the accelerator can be deployed against a live migration.

Keywords: iPaaS migration, integration intermediate representation, workflow translation, cloud-to-cloud integration, schema-agnostic translation, MuleSoft, Dell Boomi, enterprise integration migration, migration remediation reporting

1. Introduction

A platform migration in enterprise integration rarely behaves like the vendor documentation suggests it should. When an organization decides to move a set of integration flows from one iPaaS product to another, MuleSoft to Boomi, Boomi to a serverless orchestration layer such as AWS Step Functions, or any combination of the three, the assumption is often that the work is mechanical: read the source definition, write the equivalent target definition, verify, and deploy. In practice, the two platforms rarely describe the same operation the same way. A MuleSoft flow expresses its logic through Mule XML, connector configuration, and DataWeave transformations; a Boomi process is built from shapes, profiles, maps, and process properties; AWS Step Functions describes state transitions in JSON and defers most execution detail to other AWS services. Two flows that solve the same business problem can therefore look almost nothing alike once they are broken down into platform-native components.

This impedance mismatch is where migration projects lose their schedule. A conditional branch usually crosses the boundary cleanly: a MuleSoft Choice Router with two or three mutually exclusive conditions has an almost mechanical equivalent in a Boomi Decision shape, and a competent engineer converts it in minutes once the mapping pattern is known. A connector with platform-managed credentials, a retry policy tied to a specific runtime's failure-

Copyright: © 2026 by the authors

propagation model, or a parallel-branch aggregation step frequently does not convert so cleanly, and a text-to-text converter that treats all of these the same way produces target code that compiles but behaves differently under load, under failure, or under a large payload. The danger is not that the harder cases exist—every migration has some—but that a converter confident enough to guess at all of them uniformly gives a migration team no way to tell, before deployment, which parts of the result it should actually trust.

Most teams respond to this uncertainty by falling back to manual inventory and reconstruction, which holds up reasonably well for a handful of interfaces and becomes expensive once an enterprise integration portfolio spans customer, finance, human resources, and supply chain systems in the hundreds of flows. It is also inconsistent in a way that pure cost figures understate: different engineers resolve the same ambiguous construct in different ways, so a portfolio migrated by five engineers over six months rarely ends up with a single coherent mapping convention, which then becomes its maintenance liability once the migration is declared complete and the original engineers move to other work. Tooling exists for parts of this problem: connector-specific converters, vendor migration utilities, and increasingly LLM-assisted code-transformation assistants, but almost all of it optimizes for producing target artifacts quickly, not for making explicit which of those artifacts the tool is actually confident about.

We argue that a defensible middle path exists between full manual reconstruction and a naive automated converter: an explicit, inspectable integration intermediate representation that captures not just what a flow's components are but what happens to a message as it moves through them, under what conditions, and what is supposed to happen when something fails. Separating the migration into two stages, pa—parsing the source workload into this IR and then generating target-platform artifacts from the IR—es the conversion rules something a reviewing engineer can actually read, test, and argue with, instead of a black box that either works or silently does not. Just as importantly, it gives the pipeline a natural place to say "I do not know" about a specific construct, instead of forcing every component through the same automated path regardless of confidence.

This paper makes three contributions. First, we formalize an integration IR as a directed graph that treats workflow behavior branching, failure handling, and concurrency as a first-class citizen alongside data shape, extending a lineage of schema- and data-mapping research [1], [2] that has historically concentrated on data shape alone; this distinction matters because a migration that gets the data shape right but the failure behavior wrong has not actually preserved the flow's intent, only its happy path. Second, we describe a translation pipeline source parsing, normalization, semantic checking, and target generation that couples every unsupported construct to an explicit, auditable remediation report instead of a silent best-effort guess so that the artifact a migration team receives tells them exactly where their remaining manual effort needs to go, not leaving them to discover it in production. Third, we propose and demonstrate, via a transparent simulated benchmark, a validation methodology structured so that a real pilot's results would be reproducible and comparable against this baseline, not reported as an unqualified headline percentage the way migration tooling claims often are. The remainder of the paper reviews related migration and schema-translation research (Section II), formalizes the IR and translation pipeline (Section III), presents the simulated evaluation (Section IV), discusses limitations and positioning (Section V), and concludes with the case for a controlled production pilot (Section VI).

This paper does not choose the three target platforms—Dell Boomi, MuleSoft, and AWS Step Functions—arbitrarily. Boomi and MuleSoft together cover a large share of enterprise iPaaS deployments in regulated and B2B-heavy sectors such as healthcare, insurance, and logistics, where EDI trading-partner integration and platform-managed connector credentials are the norm, not the exception. AWS Step Functions represents the increasingly common target of a "move from a managed iPaaS to a cloud-native orchestration layer" migration that many of the same organizations are now considering. A translation approach validated only against a narrower slice of this space, such as two REST-centric platforms, would provide little insight into the EDI- and connector-heavy portfolios where migration effort is currently concentrated.

2. Related Work

Schema and data mapping research supplies the oldest and most directly relevant foundation for this work. Rahm and Bernstein's survey of automatic schema-matching approaches [1] catalogs the structural, linguistic, and instance-based techniques that later data-integration systems use to reconcile heterogeneous data models, while Lenzerini's theoretical treatment of data integration [2] formalizes the global-as-view and local-as-view distinctions that still underlie how integration systems reason about source-to-target correspondence. Both traditions, however, treat the integration problem primarily as a question of data shape: what does a record look like on one side, and what should it look like on the other? An iPaaS migration inherits this problem but adds a second one on top of it. What happens to the message, in what order, and under which failure conditions? Neither survey was built to address this, because neither was written with a stateful, failure-prone runtime environment in mind.

Model transformation research narrows part of that gap. Czarnecki and Helsen's feature-based survey of model transformation approaches [3] classifies transformation techniques by the structural properties they preserve and the rule languages they use, and its taxonomy of graph-based transformation strategies maps reasonably well onto the node-and-edge structure an integration flow naturally takes. Where this literature is thinner is on the workflow-execution side: retry semantics, parallel aggregation, and platform-managed credentials are not typically part of a general-purpose model-transformation taxonomy, because most of that research grew out of software and data modeling instead of runtime orchestration, and a taxonomy built for one does not automatically transfer its guarantees to the other.

Cloud interoperability standards approach the portability problem from a different angle, specifying a platform-neutral topology instead of translating between two concrete platforms after the fact. Bellendorf and Mann's survey of TOSCA-based cloud topology specification [4] documents more than a hundred papers attempting exactly this kind of standardized description, and Ramalingam and Mohan's review of semantic standards for multi-cloud portability [5] reaches a similar conclusion from the interoperability-standards side: proprietary APIs and inconsistent semantic models remain the practical barrier to seamless migration, standards efforts notwithstanding. Neither line of work, however, targets the iPaaS integration-flow layer specifically, where connector behavior and B2B trading-partner semantics add constraints a general cloud-topology standard does not model and where the practical question is rarely "Which standard should we adopt?" But "What do we do with the flows we already have, written against a platform that predates any standard we might now adopt?"

The low-code and no-code literature is the closest analogue to the platforms this paper is concerned with. Cabot's positioning of the low-code movement within model-driven engineering [6] frames low-code platforms as, in effect, model-driven engineering with the model largely hidden from the end user, a framing that explains why migrating a low-code or iPaaS flow is closer to a model-transformation problem than a code-translation one, even though the platforms rarely present themselves that way. Empirical work on legacy and cloud migration, including Gholami et al.'s field study of the practical difficulties organizations face when migrating legacy systems to the cloud [7] and Hasan et al.'s architectural review of the same space [8], repeatedly identifies the same three obstacles this paper addresses directly: undocumented or tacit platform-specific behavior, the cost of manual reconstruction at scale, and the risk of a migration that appears to succeed while silently changing behavior.

Two adjacent bodies of work inform the accelerator's design choices without directly overlapping its scope. Wen et al.'s systematic review of serverless computing research [9] and their earlier direct measurement study characterizing commodity serverless platforms [10] together show how differently a target platform such as AWS Step Functions handles concurrency, retries, and cold-start behavior compared to a stateful orchestration engine—one at the survey level and the other by measuring actual platform behavior—reinforcing why a target-generic mapping rule cannot safely assume uniform runtime semantics across targets. Parciak et al.'s experimental study of large-language-model-based schema matching [11] is the most direct evidence available on whether machine-learning techniques could eventually suggest, rather than execute, ambiguous mappings; their finding that match quality degrades with both too little and too much contextual input is a caution this paper takes seriously in treating any future ML-assisted mapping suggestion as a reviewable candidate and not an automatic decision.

This journal's own recent publication record reflects the same cluster of concerns from adjacent angles. A shared-state multi-agent framework for validating safe Java code migration [12] tackles a structurally similar problem deciding when an automatically generated migration artifact can be trusted for source-code migration instead of integration-flow migration, and its validator-veto discipline is a close cousin of the remediation-report mechanism this paper proposes at the workflow level. A monitoring framework for large-scale data-pipeline failure recovery [13] addresses the operational side of the same reliability concern; this paper's remediation report is meant to serve upstream of deployment, and a self-tuning batch advisor for low-code bulk data ingestion [14] shows that the same low-code platform family this paper targets is already accumulating its own adaptive-configuration research, independent of migration specifically.

Positioned against this literature, the gap this paper addresses is narrow but consequential: the schema- and data-integration tradition solved a version of the mapping problem without treating workflow behavior as first-class; the model-transformation and cloud-standards traditions formalize structure without engaging the platform-specific runtime differences that Wen et al.'s measurements document directly; and the low-code and migration-empirics literature diagnoses the practical cost of manual reconstruction without offering a translation mechanism that makes its own confidence visible. The integration IR and remediation-reporting pipeline described in the next section targets that intersection directly.

The EDI and B2B trading-partner layer, specifically a substantial share of the flows this paper's target platforms are used to build, adds a constraint that none of the general migration or schema-matching literature engages with directly: a mapping error in a healthcare claims transaction or a financial settlement message is not merely a bug to be triaged after the fact but a compliance event, and recent quality-engineering work on EDI gateway migration [16] treats every unverified transformation as carrying that weight from the outset. This paper's remediation-reporting mechanism is designed to generalize that discipline to iPaaS migration broadly, not only to EDI-specific flows, on the premise that a silent behavioral change is costly whether or not a regulator happens to be watching.

3. Methodology: The Integration Intermediate Representation and Translation Pipeline

A. Integration IR formalization. We represent an integration flow as a directed graph G = (V, E), where V is the set of processing nodes and E is the set of directed message paths between them. A node corresponds to an inbound listener, a connector call, a transformation, a decision point, a split, an aggregation, or an outbound operation; an edge records the movement of a message and may carry a condition, such as the outcome of a decision or the path taken on error. Building on the classification vocabulary that model-transformation research already provides [3], we treat each node type as carrying two kinds of information: a structural signature (its inputs, outputs, and control-flow role) and a behavioral annotation (its failure-handling policy, concurrency characteristics, and any platform-specific option that has no general equivalent). Platform-only options are recorded as annotations instead of being folded into the core node type, so that a target generator can choose, case by case, to map an annotation, ignore it with a warning, or flag it for manual remediation, a design decision that keeps vendor-specific detail from crowding out the shared model, consistent with how TOSCA-style topology specifications separate portable structure from platform-specific extensions [4]. Table 1 summarizes how the core IR node types correspond to concrete constructs on the three target platforms this paper considers, illustrating that the correspondence is close enough to be usable but not close enough to be automatic in every case. A Route node maps to a Boomi Decision shape with little ambiguity, while a Connector node's mapping depends entirely on whether the target platform exposes an equivalent managed connector at all.

B. Source parsing and normalization.

The pipeline begins by reading the source project and constructing an abstract syntax tree: application XML, connector configuration, and referenced transformation files for a MuleSoft source; process components, maps, profiles, and environment extensions for a Boomi source. The source analyzer's first task is to flag constructs that resist mechanical conversion: embedded Java, third-party libraries, custom connectors, and scripts that call platform runtime APIs directly because these determine a migration's true cost far more than the straightforward constructs do. The AST is then normalized into IR nodes: a MuleSoft choice router becomes a generic Route node carrying an

ordered list of conditions; a flow reference becomes a Call node pointing at another IR subgraph, later regenerated as a Boomi subprocess call or an equivalent construct on another target. Normalization is not considered complete until a semantic-checking pass confirms that every referenced subflow actually exists, every branch implied by the source is represented in the IR, data contracts are compatible at each node boundary, and no error path has been silently dropped in translation, the last of these being, in our experience, the single most consequential failure mode a converter can introduce without anyone noticing until production, because a dropped error path does not fail loudly; it fails by quietly letting an exception propagate somewhere the original flow would have caught it.

4. Mapping and target code generation.

Target generation draws on a library of mapping rules, each of which declares the IR pattern it accepts, the target-platform components it produces, and the conditions under which applying it would be unsafe. Structural mappings route nodes to Boomi Decision shapes or an equivalent choice construct. Call nodes to subprocess invocations where the source flow has a clear input/output contract are comparatively low-risk and can be tested independently of any specific source or target pair. Retry and exception nodes require more care: a rule can generate a target-platform retry construct only after comparing retry count, delay, and failure-propagation behavior between source and target, because platforms that appear to offer "the same" retry semantics frequently do not once transient-failure classification is examined closely. Parallel branches carry the same caution for a related reason: concurrency limits, ordering guarantees, and aggregation behavior vary enough between platforms [9] that a rule confident enough to run unattended on this construct class would be overconfident.

Transformation mappings separate cleanly along a similar predictability line. Simple field copies, constant assignments, arithmetic, and conditional value assignment translate through table-driven rules with little ambiguity; a rule of this kind can, for instance, take a source expression that increases a price field by a fixed percentage and regenerate it directly as an equivalent Boomi map function without engineering review because the transformation's intent and its boundary conditions (can the field be null?) Is the numeric type preserved? (Is rounding handled the same way?) are fully expressible in the rule itself. More elaborate transformation expressions, the kind of nested, conditional data Weave logic that accumulates in a mature integration portfolio are better generated as an explicit target-platform script (a Boomi Groovy or JavaScript component, for instance) selected for readability and maintainability, not merely for whether the generated code executes; a transformation that runs correctly but that no one on the receiving team can safely modify has only partially solved the migration problem. When a target construct is exposed as an API, such as a REST endpoint fronting a migrated flow, the choice of API design technique measurably affects gateway-level latency and availability and is not merely a stylistic decision [15]. In this case, the generator's connector mappings are selected with that constraint in mind, rather than defaulting uniformly to whichever style the source happened to use.

For every construct the generator cannot convert with confidence, it emits a remediation report entry naming the source component, the reason automatic conversion was withheld, and a recommended next step, such as replacing a proprietary connector, rewriting embedded code, or explicitly confirming that an observed platform-specific retry behavior is acceptable as-is. This report, and not the generated code alone, is what allows a migration team to estimate the manual work still ahead of them before regression testing begins, and it is written with the same discipline that compliance-sensitive integration domains already demand: a healthcare EDI gateway migration, for example, treats every unverified mapping as a defect risk, not a cosmetic gap [16], and the remediation report is designed to carry that same standard into domains that are not formally regulated but where a silent behavioral change is just as costly. The report is deliberately structured as a list of decisions still owed to a human reviewer, not a list of apologies for what the tool could not do, a distinction that matters for how a migration team actually uses it under deadline pressure.

5. Complexity and scalability considerations

The translation pipeline's cost is dominated by the semantic-checking pass, not by parsing or generation, because checking that every data contract is compatible across a node boundary requires comparing schemas pairwise along every edge in the IR graph instead of inspecting nodes in isolation. For a flow of moderate size, this is not a

practical bottleneck, but for an enterprise portfolio comprising hundreds of flows with shared subprocess references, the underlying graph is not really hundreds of separate small graphs but one large graph with extensive cross-references, and the semantic checker is designed to operate over that shared structure, incrementally checking only the subgraphs touched by a given migration batch, instead of re-verifying the entire portfolio on every run. This design choice has not been tested at portfolio scale; it is a stated assumption that the eventual pilot in Section VI will need to confirm, alongside the accuracy questions that the simulated evaluation in Section IV addresses.

Table 1. Integration IR node types and their target-platform mapping equivalents

IR Node Type	Semantic Role	MuleSoft Equivalent	Dell Boomi Equivalent	AWS Step Functions Equivalent
Route	Conditional branching	Choice Router	Decision shape	Choice state
Call	Subprocess invocation	Flow Reference	Subprocess (Process Call)	Nested state machine / Map state
Transform	Field-level data mapping	DataWeave transform	Map shape	Pass state + JSONPath
Connector	External system call	Connector (HTTP, DB, SFTP, etc.)	Connector shape / Connector action	Task state (Lambda / service integration)
Retry / Exception	Failure handling	Error handler / Until Successful scope	Try/Catch process shapes + exception handling	Retry / Catch fields on a state
Split / Aggregate	Parallel processing and merge	Scatter-Gather	Business Rules / flow-control split-join	Parallel state + ResultPath aggregation
Annotation	Platform-specific configuration	Custom properties / metadata	Process properties / extensions	State-machine definition metadata

Figure 1. Two-stage translation pipeline: source parsing and normalization into the Integration IR, followed by semantic checking and target generation with an explicit remediation-report path.

6. Evaluation

We evaluate the translation pipeline's automation coverage through a deliberately labeled simulated benchmark and not a production deployment. No executed migration data exists yet for this accelerator; the results below describe a reproducible synthetic exercise designed to test whether the category-level automation assumptions built into the mapping-rule library above are internally consistent, not to claim a measured production outcome. We describe them this way throughout and treat a live pilot, not this exercise, as the evidence that would ultimately determine production readiness, consistent with how empirical migration studies in this space are usually validated [7], [8], [17].

A. Simulated corpus construction. We constructed a representative synthetic corpus of 419 integration-flow components, distributed across the seven construct categories that the methodology section treats differently: basic routing and decision logic, simple field-level transformations, complex transformation expressions, reusable subprocess references, retry and exception-handling nodes, parallel and aggregation branches, and custom connectors or embedded code. The category proportions were set to approximate the composition of a typical enterprise integration portfolio, weighted toward routing and simple transformation work, with custom connectors and parallel branches making up a smaller minority consistent with the corpus-composition assumptions used in adjacent empirical studies of business-process-model collections [17]. Each category was assigned an automation-rate assumption directly derived from the risk classification argued for in Section III (for example, a low assumed automation rate for

custom connectors and a high one for basic routing) and not fitted after the fact to produce a favorable aggregate number. The construction procedure is deterministic and fully specified: category proportions and per-category rates are fixed inputs, component counts are the proportion applied to the corpus size, and automated/flagged counts are the corresponding share of each category's count so that the entire exercise can be regenerated exactly from the parameters stated here.

B. Results. Table 1 reports the per-category corpus composition, assumed automation rate, and estimated manual-versus-tool-assisted effort under this simulation. Across the full corpus, 68.5% of components were classified as cleanly automatable, and the remainder were flagged for the remediation-report path described in Section III-C. The estimated effort reduction, computed from the same per-category hour assumptions used in the recent mainframe-migration effort-estimation literature [18], was 69.9% relative to a fully manual reconstruction baseline. Figure 2 presents the same result as a per-category comparison, making visible what the aggregate figure conceals: automation concentrates heavily in routing, simple transformation, and subprocess-reference constructs, while custom connectors and parallel branches remain predominantly a manual-review activity under this methodology, exactly as the risk classification in Section III predicted, not as an unexpected finding.

The category-level pattern is worth reading in more detail than the aggregate figure invites. Basic routing and simple transformation together account for slightly under half of the simulated corpus and clearly well above four-fifths automation, which, under this methodology's assumptions, is exactly the outcome the structural-mapping argument in Section III-C predicts: these constructs have a close, low-ambiguity target-platform equivalent and little reason to require review. Complex transformation expressions and retry/exception-handling nodes sit in the middle of the distribution, with roughly half of each category automated and half flagged as a pattern consistent with the argument that these constructs are automatable in the common case but genuinely ambiguous often enough that a rule confident enough to skip review on all of them would be wrong a meaningful fraction of the time. Custom connectors and embedded code sit at the opposite extreme from routing and simple transformation, with automation in the single digits under this simulation's assumptions; that a graph-based IR gains almost nothing in this category is not a weakness specific to this design so much as a restatement of the underlying problem: a connector's proprietary runtime behavior is, by construction, not visible to any static analysis of the source project, simulated or real. Purely as a point of comparison, and not as a validation of either result, this simulation's aggregate automation figure lands close to the executed success rate reported for a comparably scoped Java-migration multi-agent framework published in this same venue [12]; the two figures measure different systems on different corpora through different methods, and neither confirms the other, but the proximity is at least consistent with the more general claim that roughly two-thirds automation is a plausible order of magnitude for structurally similar migration-assistance tools once custom and platform-specific constructs are excluded from the easy cases.

C. Threats to the simulation's validity. This exercise tests internal consistency, not real-world accuracy, and three limitations should be read as a companion to, not a footnote on, the results above. The category proportions and automation-rate assumptions are author-specified instead of measured from a real migration portfolio, so the aggregate figures should not be read as a prediction of what any specific organization would observe; a portfolio with an unusually high share of custom connectors, for instance, would see a materially lower aggregate automation rate than the one reported here regardless of how sound the underlying mapping rules are. The effort-hour assumptions per category are illustrative and drawn from comparable published effort-estimation work [18] instead of from timed observation of the accelerator itself, since the accelerator has not yet been run against a real codebase, and effort estimation for migration work is itself known to vary widely by organization and toolchain. Finally, a simulated corpus cannot exercise the semantic-checking pass described in Section III-B against real platform-specific edge cases or undocumented connector behavior, in particular, that only a live source system can surface, which is precisely the category of risk a synthetic exercise is least equipped to evaluate. Any eventual production pilot would still need to satisfy the same pre-deployment discipline the broader software-engineering literature insists on for validation gates generally: unit and contract tests and a systematic regression-testing pass whose practical applicability, not merely its theoretical coverage, has itself been questioned closely enough to warrant its own dedicated review [19], [20]. We

treat that gap between simulated internal consistency and validated real-world accuracy as the paper's central open problem, not as a limitation to be minimized in passing.

A secondary question this simulation can speak to, even under its stated limitations, is how sensitive the aggregate automation figure is to the corpus's category mix and not to the mapping rules themselves. Re-weighting the same seven per-category automation rates against a hypothetical portfolio with double the share of custom connectors and parallel branches, a plausible profile for an older, heavily customized integration estate, as opposed to a newer, template-driven one, would pull the aggregate automation rate down by roughly ten to fifteen percentage points under this methodology's assumptions, without any change to the mapping-rule library itself. That sensitivity is, if anything, the more actionable finding than the headline aggregate: it says that the single number a migration sponsor is most likely to ask for up front is largely a property of which portfolio is being migrated, not of how capable the tool is, and that a credible answer to "what automation rate should we expect" has to start with a category-level inventory of the specific portfolio in question, not with this or any other paper's aggregate figure.

Table 2. Simulated benchmark results by flow-component category (synthetic corpus, n = 419)

Category	Count	Automated	Flagged for remediation	Automation rate	Manual hrs (est.)	Tool-assisted hrs (est.)
Basic routing / decision logic	92	84	8	91.3%	138.0	4.1
Simple field-level transformations	109	94	15	86.2%	109.0	5.8
Complex transformation expressions	67	29	38	43.3%	268.0	76.2
Reusable subprocess / call references	59	47	12	79.7%	147.5	11.4
Retry and exception-handling nodes	42	22	20	52.4%	126.0	32.6
Parallel / aggregation branches	29	9	20	31.0%	145.0	68.3
Custom connectors and embedded code	21	2	19	9.5%	168.0	133.2
Total	419	287	132	68.5%	1101.5	331.7

Figure 2. Simulated automation rates and effort reductions by flow-component category.

7. Discussion

The central design choice this paper argues for is separating predictable structural and transformation mappings from constructs that require engineering review, and reporting the latter explicitly instead of guessing holds up against the related work reviewed in Section II in a specific way: schema-matching and data-integration research [1], [2] solved a version of this problem for data shape decades ago by treating ambiguous matches as candidates for

confirmation instead of committing to them silently, and the remediation-report mechanism here is the same idea applied to workflow behavior instead of data structure. Where this paper's approach differs most from the cloud-portability-standards literature [4], [5] is in accepting the reality, instead of trying to eliminate it, that some fraction of any real migration will not be automatable; a standard that assumes full portability is achievable tends to underserve exactly the custom-connector and embedded-code cases that our category-level results suggest carry the least automation headroom. For a migration team, the practical implication is less about which tool to buy and more about where to spend review time first: the simulated results suggest budgeting review effort in rough proportion to a portfolio's share of custom connectors and parallel branches instead of spreading review evenly across every flow, since routing and simple-transformation constructs consume review time disproportionately to the risk they actually carry.

Limitations. Beyond the simulation-specific caveats already noted in Section IV-C, the accelerator's design carries limitations that a synthetic exercise cannot surface but a production pilot would. Proprietary connector behavior is, by definition, only partially visible from the outside; the IR's annotation mechanism can record that a connector exists and record its declared configuration, but it cannot guarantee that the annotation captures every runtime nuance a vendor has not documented, and a connector's undocumented retry-on-throttling behavior is a realistic example of exactly the kind of detail that would only surface once a real workload runs against it. Performance is a related concern: a generated transformation script that passes a functional test may still run measurably slower than a hand-tuned native mapping component, and differences in transaction-boundary and retry behavior between platforms [9], [10] can matter more to a production system than a modest latency regression would. Compliance-sensitive integration domains add a further constraint the accelerator's current design does not yet formalize: a healthcare EDI trading-partner flow or a financial transaction pipeline carries regulatory and audit obligations that a generic remediation report does not automatically satisfy, even when the underlying mapping is technically correct, and extending the remediation report's vocabulary to capture domain-specific compliance flags is a natural next step, not a solved part of the current design. Neither limitation is unique to this accelerator, but neither should be minimized just because the underlying design is sound in the cases it handles well.

A further practical implication follows from the simulated results' placement of the least automatable categories. Because custom connectors and embedded code cluster at the low end of automation regardless of how the surrounding mapping-rule library is tuned, a migration team's planning conversation is better anchored on an early, targeted inventory of exactly these two categories: which proprietary connectors this portfolio actually uses and how much embedded scripting logic exists outside the platform's declarative constructs, rather than on a single portfolio-wide automation percentage that a generic accelerator, this one included, cannot promise until it has seen the real source project. These are the two limitations most likely to determine whether a real pilot's results resemble this simulation's assumptions or diverge from them, and both are squarely within scope for the pilot design proposed in Section VI, not a separate research effort.

8. Conclusion

Migrating an integration workload between iPaaS platforms is difficult for a structural reason, not an incidental one: platforms that solve the same business problems encode those solutions through genuinely different workflow models, connector behaviors, and failure semantics, and no amount of surface-level syntactic conversion resolves a mismatch that runs that deep. This paper has argued that a platform-neutral integration IR, coupled to a translation pipeline that separates predictable mappings from constructs requiring engineering review and reports the latter explicitly, is a more defensible response to that mismatch than either full manual reconstruction or a converter that guesses silently. Formalizing workflow behavior branching, failure handling, and concurrency as a first-class part of the IR, instead of an afterthought layered on top of a data-shape mapping, is what allows the translation pipeline to say plainly where it is confident and where it is not, which is the property that matters most to a migration team deciding how to allocate its remaining manual effort.

The simulated benchmark reported here is offered strictly as evidence that the underlying category-level risk assumptions are internally consistent, not as a substitute for the real pilot this approach still requires: a representative sample of live flows migrated by the accelerator and independently by hand, evaluated against the same functional, performance, and effort criteria, with failures reported as openly as successes. That pilot would also be the appropriate venue to test the compliance-reporting extension and the performance concerns raised in Section V, neither of which a synthetic corpus can meaningfully exercise. Until that pilot has been run, the honest claim this paper can make is a narrower one: the risk classification the accelerator's mapping-rule library encodes is internally consistent under simulation; the remediation-reporting mechanism gives a migration team a concrete artifact to plan around; and the next necessary step, not an optional follow-on, is measuring whether that consistency survives contact with a real, messy, undocumented production integration portfolio.

For a practitioner deciding whether to invest in a tool of this kind instead of continuing with manual reconstruction, the honest answer this paper can offer is conditional, not unqualified: where a portfolio's category mix skews toward routing, simple transformation, and well-contracted subprocess references, an IR-based accelerator of this design should meaningfully reduce reconstruction effort; where it skews toward custom connectors and undocumented embedded logic, the same accelerator's contribution is mainly to identify that risk early and explicitly, not to remove it. Both outcomes are useful to know before a migration begins, not after it has already consumed a budget and a schedule.

9. CRediT Author Contributions

Gowrunaidu Seelanki: Conceptualization, Methodology, Investigation, Writing – Original Draft, Writing – Review & Editing, Visualization.

10. Conflict of Interest

The author declares no conflict of interest.

11. Statement of Human and Animal Rights

This study did not involve human participants, animal subjects, or identifiable personal data requiring ethics board review.

12. Informed Consent

Not applicable.

13. Data Availability

The synthetic benchmark corpus and category-level assumptions described in Section IV are generated by a deterministic procedure fully specified in the text; the generating parameters are reproducible from the description given and are available from the author on request.

14. Acknowledgments

The author used Claude (Anthropic, claude-sonnet model) to assist with manuscript drafting and language refinement. The author takes full responsibility for the accuracy, originality, and integrity of all content presented in this manuscript, and confirms that all data, analysis, and conclusions reflect independent work and verified sources.

REFERENCES

1. E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," The VLDB Journal, vol. 10, 2001, pp. 334-350. Available: https://doi.org/10.1007/s007780100057
2. M. Lenzerini, "Data integration: a theoretical perspective," in Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems (PODS), 2002, pp. 233-246. Available: https://doi.org/10.1145/543613.543644

3. K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," IBM Systems Journal, vol. 45, no. 3, 2006, pp. 621-645. Available: https://doi.org/10.1147/sj.453.0621
4. J. Bellendorf and Z. A. Mann, "Specification of cloud topologies and orchestration using TOSCA: a survey," Computing, vol. 102, no. 8, 2020, pp. 1793-1815. Available: https://doi.org/10.1007/s00607-019-00750-3
5. C. Ramalingam and P. Mohan, "Addressing semantics standards for cloud portability and interoperability in multi cloud environment," Symmetry, vol. 13, no. 2, art. 317, 2021. Available: https://doi.org/10.3390/sym13020317
6. J. Cabot, "Positioning of the low-code movement within the field of model-driven engineering," in Proc. ACM/IEEE 23rd Int. Conf. Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS-C), 2020, pp. 1-3. Available: https://doi.org/10.1145/3417990.3420201
7. M. F. Gholami, F. Daneshgar, G. Beydoun, and F. Rabhi, "Challenges in migrating legacy software systems to the cloud - an empirical study," Information Systems, vol. 67, 2017, pp. 100-113. Available: https://doi.org/10.1016/j.is.2017.03.008
8. M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "Legacy systems to cloud migration: a review from the architectural perspective," Journal of Systems and Software, vol. 202, art. 111702, 2023. Available: https://doi.org/10.1016/j.jss.2023.111702
9. J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: a systematic review," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 5, art. 131, 2023. Available: https://doi.org/10.1145/3579643
10. J. Wen, Y. Liu, Z. Chen, J. Chen, and Y. Ma, "Characterizing commodity serverless computing platforms," Journal of Software: Evolution and Process, 2021, art. e2394. Available: https://doi.org/10.1002/smr.2394
11. M. Parciak, B. Vandevoort, F. Neven, L. M. Peeters, and S. Vansummeren, "Schema matching with large language models: an experimental study," in Proc. VLDB 2024 Workshop on Tabular Data Analysis (TaDA), 2024.
12. N. K. S. Beshane, "TriArchitect: a shared-state multi-agent framework for safe Java code migration," International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM), vol. 18, no. 18s, 2026, pp. 739-751. Available: https://doi.org/10.70917/ijcisim-2026-4917
13. G. R. Pappula, "A practical framework for monitoring large-scale data pipelines and recovering from failures," IJCISIM, vol. 18, no. 18s, 2026, pp. 194-203. Available: https://doi.org/10.70917/ijcisim-2026-4861
14. A. Garg and S. Patel, "HBA: a heuristic batch advisor for self-tuning bulk data ingestion in the OutSystems low-code platform," IJCISIM, vol. 18, no. 18s, 2026, pp. 329-339. Available: https://doi.org/10.70917/ijcisim-2026-4873
15. F. Aydemir and F. Bascifci, "Performance and availability analysis of API design techniques for API gateways," Arabian Journal for Science and Engineering, vol. 50, no. 15, 2025, pp. 11485-11498. Available: https://doi.org/10.1007/s13369-024-09474-9
16. D. Manoharan, "Advancing healthcare EDI interoperability through Informatica Cloud B2B Gateway quality engineering," International Journal of Computational and Experimental Science and Engineering, vol. 12, no. 1, 2026. Available: https://doi.org/10.22399/ijcesen.4925
17. M. Saeedi Nikoo, S. Kochanthara, O. Babur, and M. van den Brand, "An empirical study of business process models and model clones on GitHub," Empirical Software Engineering, vol. 30, no. 2, art. 48, 2025. Available: https://doi.org/10.1007/s10664-024-10584-z
18. S. Roth, "Effort estimation of large-scale enterprise application mainframe migration projects: a case study," in Proc. 27th Int. Conf. Enterprise Information Systems (ICEIS), vol. 2, 2025, pp. 845-850.
19. R. Greca, B. Miranda, and A. Bertolino, "State of practical applicability of regression testing research: a live systematic literature review," ACM Computing Surveys, vol. 55, no. 13s, 2023. Available: https://doi.org/10.1145/3579851
20. S. Das and K. Gary, "Regression testing in agile - a systematic mapping study," Software, vol. 4, no. 2, art. 9, 2025. Available: https://doi.org/10.3390/software4020009