Journal of Computing and Data Technology

https://journal.krrishpub.com/index.php/jcdt/index

RESEARCH ARTICLE

Integration Debt in Multi-Vendor Enterprise Estates: Accumulation Mechanisms and an Assessment Approach

Anil Chandra Rajendra Pallikonda [a,*]

[a] *Integration Architect, Blue Ribbon Global Technologies LLC, Omaha, Nebraska 68131, United States of America.*

Abstract

Technical debt is well characterized within application codebases, where it can often be attributed to an owning team and repaid within a single deployment boundary. Integration debt behaves differently. It accumulates between independently governed systems, where release schedules, commercial ownership, platform choices, and semantic definitions are distributed across organizations. This paper treats integration debt as a distinct category and characterizes four accumulation mechanisms in multi-vendor enterprise estates: point-to-point proliferation, semantic divergence, orphaned interfaces, and platform sprawl. We extend the original qualitative assessment with measurable indicators for coupling breadth, specification recoverability, consumer volatility, and semantic divergence, and represent remediation priority as a two-dimensional coordination–knowledge exposure rather than an unsupported scalar debt score. To ground the method in observable interface data, we add a reproducible snapshot of the public APIs.guru OpenAPI directory, whose current metrics expose 3,992 specifications across 677 providers together with validity, reachability, repair, and weekly-change signals. Executable Python code is provided to refresh the snapshot. The resulting framework is intended to sequence remediation: opaque and semantically divergent dependencies are addressed before high-volume but well-understood interfaces, because knowledge loss, unlike coordination cost, can make later modernization effectively unbounded.

Keywords: Integration debt, Technical debt, Enterprise integration, Legacy modernization, Software architecture.

Article information:
Published by: **Krrish Scientific Publications Pvt. Ltd.**
DOI: https://doi.org/10.71426/jcdt.v1.si1.pp10-18
Received: *09 Sep. 2025* | Revised: *30 Sep. 2025* | Accepted: *08 Oct. 2025* | Published: *16 Oct. 2025*
Copyright ©2025 Author(s).
This is an open-access article distributed under the terms of the Creative Commons Attribution–NonCommercial 4.0 International License (CC BY-NC 4.0).

1. Introduction

Enterprise integration is the discipline of making independently conceived systems exchange data coherently. It is also the discipline in which the consequences of expedient decisions persist longest. An interface built in a week to meet a delivery date may remain in production for a decade, carrying assumptions its authors never documented, consumed by systems that did not exist when it was written, and owned by a team that has since been reorganized.

The software engineering community has developed a rich vocabulary for this phenomenon inside application boundaries. Technical debt has been systematically mapped [1], its architectural variant modeled and reviewed [2], its identification techniques surveyed [3], and its accumulation and refactoring observed longitudinally across industrial cases [4]. That body of work shares a structural assumption: that the indebted artifact has an owner who can decide to repay it.

Integration debt violates that assumption. The indebted artifact is an interface, and an interface is a shared object. Its producer and consumer may sit in different divisions, report through different management chains, run different release calendars, and in the multi-vendor case be bound by different commercial agreements. No party can unilaterally repay the debt, because repayment requires coordinated change on both sides of a contract neither side fully controls.

This distinction is not merely organizational bookkeeping. It changes which remediation strategies are available. Incremental refactoring, the standard response to code debt, presumes that a change can be made and validated within one deployment unit. Interface refactoring cannot be validated that way: the consequences appear in a system the refactoring team does not operate, often only under production load, and often after a delay. The practical result is that integration debt is repaid rarely, and when it is repaid, it is repaid through large coordinated programmes

*Corresponding author
Email address: rajendra.pac@gmail.com (Anil Chandra Rajendra Pallikonda).

rather than continuous improvement.

2. Related work

2.1. Technical debt

The technical debt literature is mature. Systematic mappings have catalogued debt types, identification indicators, and management activities [1], [5]. Architectural technical debt has received dedicated treatment, including a unified model synthesizing the prior literature [2] and a mapping of identification techniques and their readiness for industrial adoption [3]. Longitudinal case work has traced how architectural debt accumulates and is refactored over time in industrial settings [4]. The debt metaphor has also been formalized beyond rhetoric as a decision problem involving short-term benefit and later change cost [18]; empirical work on principal and interest shows that the relation between the two is not simple enough to support naive monetary ranking [19]. A systematic review of prioritization approaches similarly finds limited consensus on which factors should determine repayment order [20]. These results motivate our use of a multi-property ordering rather than a single financial debt score.

2.2. Integration and service-oriented architecture

Work on enterprise integration has largely concerned construction mechanism rather than portfolio degradation. Composition and mediation on the enterprise service bus have been formalized and automated [6]. Analyses of service-oriented architecture in variability-intensive environments identify recurring pitfalls in reuse-oriented design [7], while distributed middleware research emphasizes designing for change across heterogeneous environments [8]. This literature explains how integrations can be constructed and evolved; it says comparatively little about what happens when responsibility is spread across vendors and the interface portfolio persists for a decade or more.

2.3. Fragmentation and its consequences

More recent work documents the conditions under which integration complexity grows. Surveys of microservices principles and migration challenges record architectural fragmentation that propagates into integration complexity [9]. Grey-literature synthesis identifies maintenance burden as a recurring reported cost of microservices [10]. Enterprise architecture management research documents the difficulty of maintaining an accurate view of a fragmented landscape [11], which is a precondition for the orphaning mechanism described later. A systematic review of multi-cloud native applications reports heterogeneity of architectures, methods, and tools as an open problem [12]. Large-scale dataflow work shows why streaming semantics, event time, and evolving state are not reducible to ordinary request–response integration [13], making some platform plurality structurally necessary.

2.4. Interface lifecycle and API evolution

API-management research is the closest adjacent body of work. A focus-area maturity model decomposes API management into lifecycle, security, performance, observability, community, and commercial concerns [14], while integration-platform research finds strategy still less developed than technology selection [15]. Empirical study of Web API evolution shows that nominally loosely coupled clients remain strongly affected by provider-side changes [21]. Interface-evolution patterns therefore emphasize explicit choices between compatibility and extensibility over the service lifecycle [22]. API-deprecation research identifies communication with clients as a central lifecycle concern and notes that remote APIs remain comparatively understudied. Usage-driven evolution work further shows that consumer logs can be used to infer the effects and needs that should guide API change. Together, these studies support consumer volatility and recoverability as first-class estate properties.

2.5. Semantics and data quality

Schema matching has been surveyed as a problem of reconciling structurally different representations of overlapping data [16]. Data-quality research established that quality is multi-dimensional and defined relative to consumer fitness for use rather than intrinsic correctness [17]. We adopt this consumer-relative framing when arguing that semantic divergence cannot be "repaired" merely by inserting another transformation.

Existing technical-debt work is strongest inside software-system boundaries; interface-lifecycle work concentrates on provider–client evolution; integration research concentrates on construction. None directly models the combined accumulation of distributed ownership, knowledge loss, semantic divergence, and platform residue across a multi-vendor interface estate. This paper addresses that gap.

3. Integration debt as a distinct category

We define integration debt as the accumulated future cost, borne across multiple systems, of interface decisions taken without reference to the interface portfolio as a whole.

Three properties distinguish it from architectural technical debt as conventionally modeled [2].

i) Distributed ownership

A code debt item sits inside a deployment unit with an owning team. An interface has at least two owners and frequently more, since a widely consumed interface acquires owners as consumers attach to it. Repayment therefore requires negotiated rather than delegated authority.

ii) Deferred and displaced consequence

The cost of code debt is usually borne by the team that incurred it, in the form of slower subsequent change. The cost of integration debt is typically borne by a different team at a later time: the producer takes the shortcut and the consumer absorbs the breakage, often during an unrelated upgrade years later.

iii) Specification loss

Code carries its own specification, in that the source is the authoritative statement of behaviour. An interface's

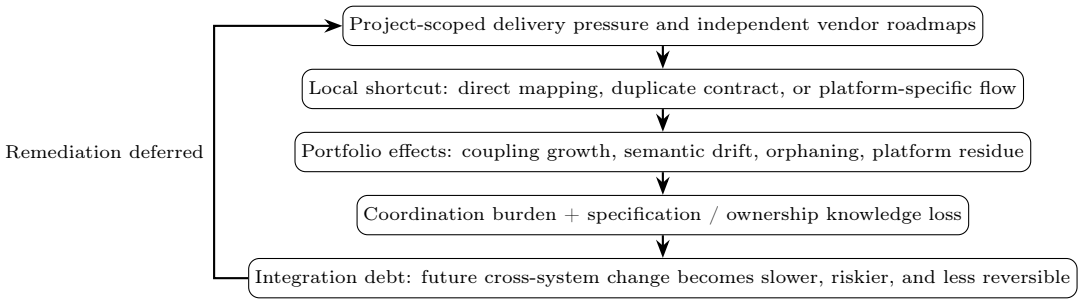

Figure 1: Accumulation loop for integration debt in multi-vendor estates.

Table 1: Code debt and integration debt contrasted.

Property	Code debt	Integration debt
Ownership	Single team	Two or more parties
Cost bearer	Usually incurrer	Usually another party
Timing	Near term	Deferred, often years
Specification	Source is truth	Often only in the flow
Repayment unit	Deployment unit	Coordinated programme
Validation	Local tests	Cross-system, in production

behavioural contract frequently exists only in the transformation logic of the integration itself, which is why integration estates so often contain the only surviving statement of a business rule. This property has no clean analogue in code debt and is central to the assessment approach we propose.

Table 1 summarizes the contrast.

4. Accumulation mechanisms

We characterize four mechanisms. Each is stated with the organizational condition that produces it, the reason conventional remediation fails, and the observable trace it leaves in the estate.

4.1. M1: Point-to-Point proliferation under delivery pressure

The most familiar mechanism is the direct interface built because it is the fastest path to a delivery date. Each individual decision is locally rational: a point-to-point interface between two known systems is cheaper to build, cheaper to test, and carries no dependency on a shared abstraction that does not yet exist.

The condition that produces it is not ignorance but incentive structure. Delivery accountability is scoped to a project, while the cost of proliferation is borne by the portfolio. A project manager who invests in an abstraction pays the full cost and captures a fraction of the benefit, with the remainder accruing to projects not yet chartered.

Conventional remediation fails because the resulting mesh has no natural decomposition point. Refactoring any single edge requires the agreement of both endpoints, and the benefit of refactoring one edge in a mesh of n edges is approximately $1/n$ of the total available benefit, while the coordination cost is fixed. Remediation is therefore rational only in bulk, which places it beyond the funding horizon of ordinary maintenance.

The observable trace is a high ratio of interfaces to systems, with interfaces concentrated on a small number of heavily consumed source systems.

4.2. M2: Semantic divergence

Two systems may both hold a *customer* record and disagree about what a customer is: whether a prospect qualifies, whether a terminated account persists, whether a household is one customer or several. Integration surfaces the disagreement without resolving it.

The condition is organizational rather than technical. Definitions diverge because the business functions that own them have different purposes, and each definition is correct for its owner. The schema matching literature treats reconciliation as an inference problem over representations [16]; the data quality literature is closer to the mark in framing quality as fitness for a consumer's use rather than intrinsic correctness [17]. Where two consumers require incompatible fitness, no transformation is correct for both.

Remediation in the integration layer fails by construction. A transformation that reconciles two definitions encodes a third definition, which is authoritative nowhere and maintained by a team that owns neither source. Each subsequent consumer either adopts this unofficial definition or adds another, and the estate accumulates what may be termed semantic debt: a growing population of transformation rules that constitute an undocumented and uncoordinated data model.

The observable trace is transformation logic of disproportionate complexity relative to the structural difference between the schemas it bridges.

4.3. M3: Orphaned interfaces

Interfaces outlive their consumers. A downstream system is decommissioned, replaced, or quietly stops reading, and the interface that fed it continues to execute, consuming capacity, appearing in dependency analyses, and constraining change to its source.

The condition is asymmetric visibility. Decommissioning a system is a project with a plan; discovering every interface that fed it is a discovery exercise that the decommissioning project has no incentive to fund. Enterprise architecture management research documents precisely this difficulty of maintaining an accurate landscape view in fragmented estates [11].

Remediation fails for an unusual reason: the cost of verifying that an interface is genuinely unused frequently exceeds the cost of leaving it running. Absence of traffic is

Table 2: Accumulation mechanisms and their observable traces

Mech.	Producing condition	Observable trace
M1	Project-scoped delivery accountability	High interface-to-system ratio, concentrated on few sources
M2	Divergent business definitions with separate owners	Transformation complexity out of proportion to schema difference
M3	Asymmetric visibility at decommissioning	Interfaces with no consumer contact, consumers absent from inventory
M4	Per-project platform selection	Same pattern on multiple platforms, platforms held for a residue

not proof of obsolescence, since the interface may serve a quarterly or annual process, and the party who would benefit from removal is rarely the party who would be blamed for a wrongful removal. The asymmetry of consequence favours inaction.

The observable trace is interfaces with no recorded consumer contact over an extended window, and interfaces whose registered consumers no longer appear in the application inventory.

4.4. M4: Platform sprawl

Enterprises accumulate integration platforms. A general-purpose integration platform is licensed; an ERP or CRM vendor bundles a native integration layer; event streaming infrastructure is introduced for a real-time programme; a cloud provider's native integration services arrive with a migration. Each was justified. Collectively they present overlapping capability with no allocation rule.

The condition is that platform selection is made per project, at the moment of need, by teams optimizing for immediate delivery and existing skills. The literature on integration platform selection finds precisely this: choice driven by a combination of technical fit, organizational familiarity and cost [15]. Structural forces reinforce it, since streaming and request-response paradigms have resisted unification under a single abstraction [13], so some plurality is unavoidable and the boundary between necessary and accidental plurality is never drawn.

Remediation fails because consolidation is framed as a platform decision when it is a portfolio decision. Migrating interfaces from one platform to another delivers benefit only if the source platform is subsequently retired, which requires migrating every interface on it, including those whose specification is no longer recoverable.

The observable trace is the same integration pattern implemented on multiple platforms within one estate, and platforms retained for a small residue of interfaces.

5. An assessment approach

Making integration debt visible requires properties that can be assessed from artifacts organizations already hold, since an assessment requiring new instrumentation will not be performed on the estates that most need it. We propose four.

i) P1: Coupling breadth

For a given interface, the number of distinct systems whose correct operation depends on its behaviour, counting transitively where an interface feeds a system that other interfaces read.

Coupling breadth estimates the coordination cost of change. It is assessable from interface inventories and dependency documentation, imperfectly but sufficiently for ranking. It is the dominant term in remediation cost, because coordination cost scales with the number of parties who must agree rather than with the technical difficulty of the change.

ii) P2: Specification recoverability

The degree to which the intended behaviour of an interface can be reconstructed from available artifacts without recourse to the people who built it.

We distinguish three levels. *Documented*: a current specification exists and matches observed behaviour. *Derivable*: no specification exists, but behaviour can be reconstructed from readable transformation logic within reasonable effort. *Opaque*: behaviour is encoded in logic that is unreadable, generated, or dependent on runtime configuration not held with the artifact.

Specification recoverability is the property most often omitted from technical debt models, because in code debt the source is the specification. In integration estates it is frequently the binding constraint on remediation: an opaque interface cannot be safely replaced at any price, only reverse-engineered from observed traffic and then replaced, which is a materially different and more expensive undertaking.

iii) P3: Consumer volatility

The rate at which the set of consumers of an interface changes, measured over whatever window the organization's records support.

Volatility predicts where abstraction investment pays. An interface with a stable consumer set for years is a poor candidate for abstraction regardless of its coupling breadth, because the abstraction's benefit accrues on consumer change. An interface acquiring consumers is a strong candidate even at low current breadth, because it is on the path to becoming a bottleneck.

iv) P4: Semantic divergence

The number of distinct definitions of a business entity in active use across the systems an interface connects, and whether the interface transformation reconciles them or propagates them.

This is the hardest property to assess and the most consequential, since it determines whether debt is repayable in the integration layer at all. Divergence must be resolved in data governance, through canonical definitions with organizational ownership [17]; an interface can only expose it. An estate scoring poorly here is one where integration remediation alone will not succeed.

5.1. Operationalization and composition

The properties are intentionally retained as separate dimensions, but they can still be measured consistently. For interface i, let $\mathcal{C}_i(t)$ be the set of direct and transitive consumers known at time t, D_i the number of materially distinct business definitions encountered across connected

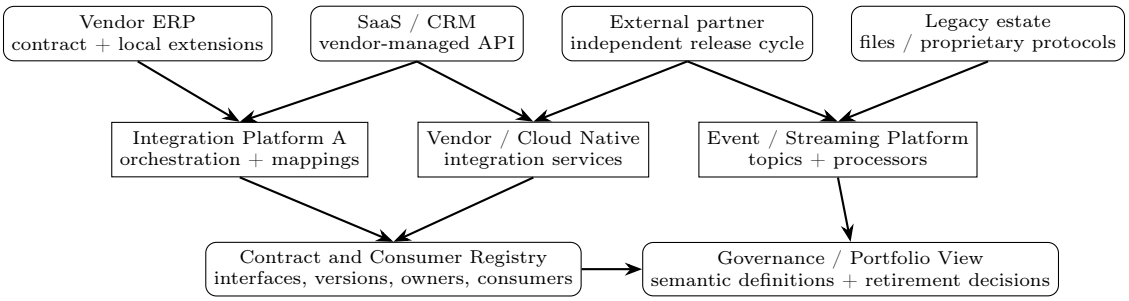

Figure 2: Multi-vendor estate model.

systems, and $D_{\max}$ the largest such count in the assessed scope. We define normalized coupling breadth (1):

$$b_i = \frac{\log\left(1 + |\mathcal{C}_i(t)|\right)}{\log\left(1 + B_{\max}\right)}, \tag{1}$$

In (1), $B_{\max}$ is the maximum observed consumer breadth. Log scaling prevents one exceptionally popular interface from flattening the rest of the estate.

Specification recoverability is represented as a risk level rather than as a quality score:

$$r_i = \begin{cases} 0, & \text{documented and behaviorally current,} \\ 0.5, & \text{derivable from readable} \\ & \text{implementation artifacts,} \\ 1, & \text{opaque or dependent on unavailable} \\ & \text{runtime knowledge.} \end{cases} \tag{2}$$

Consumer volatility over assessment window Δ is given by (3).

$$v_i = \min\left(1, \frac{|\mathcal{C}_i(t) \triangle \mathcal{C}_i(t-\Delta)|}{\max\left(1, |\mathcal{C}_i(t-\Delta)|\right)}\right), \tag{3}$$

In (3), $\triangle$ denotes symmetric set difference. Semantic divergence is normalized as (4).

$$d_i = \frac{\max(0, D_i - 1)}{\max(1, D_{\max} - 1)}. \tag{4}$$

We deliberately do not collapse these measures into one debt index. Instead, two exposure coordinates preserve the distinction that matters operationally:

$$E_i^{\text{coord}} = \sqrt{b_i v_i}, \tag{5}$$

$$E_i^{\text{know}} = \sqrt{r_i d_i}. \tag{6}$$

The first captures coordination pressure generated by broad and changing consumer populations; the second captures the interaction of lost specification with incompatible meanings. Remediation can then be sequenced lexicographically by (7).

$$\Pi(i) = \left(\mathbb{1}[r_i = 1], E_i^{\text{know}}, E_i^{\text{coord}}\right), \tag{7}$$

Equation (7) is an ordering rule, not an empirically calibrated cost model.

The four profiles are not a classification of whole enterprises but of interfaces within an estate. A large estate will contain all four. The value of the distinction is sequencing: interfaces in the *knowledge* profile must be addressed before *coordination*-profile interfaces that depend on them, because a coordination programme that encounters an opaque dependency mid-flight loses the reversibility on which incremental modernization depends.

Table 3: Estate profiles and implied remediation posture

Profile	P1	P2	P3	P4	Posture
Contained	Low	High	Low	Low	Routine maintenance
Coordination	High	High	High	Low	Abstraction investment
Knowledge	Low	Low	Low	Med	Recover before change
Structural	High	Low	High	High	Governance first

5.2. Reproducible public interface snapshot

A public directory cannot reproduce the governance conditions of a private enterprise estate, but it can provide a non-fabricated interface corpus for illustrating the type of metadata signals the assessment should ingest. We use the APIs.guru metrics endpoint, a continuously maintained directory of OpenAPI descriptions. At the snapshot used for this manuscript, the endpoint reports 3,992 specifications, 2,529 API descriptions, 108,837 endpoints, and 677 providers. It also reports 688 invalid specifications, 166 unreachable specifications, 84,860 applied fixes, and 437 specifications updated during the reported week.[1]

The provider distribution is also materially concentrated. The largest named provider contributes 1,829 specifications (45.82% of the directory), followed by 464 from `googleapis.com`, 286 from `amazonaws.com`, and 181 from `apisetu.gov.in`; together these four named providers account for 69.14% of all specifications in the snapshot. This is not presented as an estimate of enterprise vendor concentration. It is an observable example of why interface counts alone are insufficient: portfolio shape, provider concentration, validity, reachability, and change rate are distinct signals.

Listing 1: Refresh the public snapshot and derive concentration / integrity indicators

[1]Public metrics endpoint: https://api.apis.guru/v2/metrics.json. The endpoint is live, so values can change after manuscript preparation.

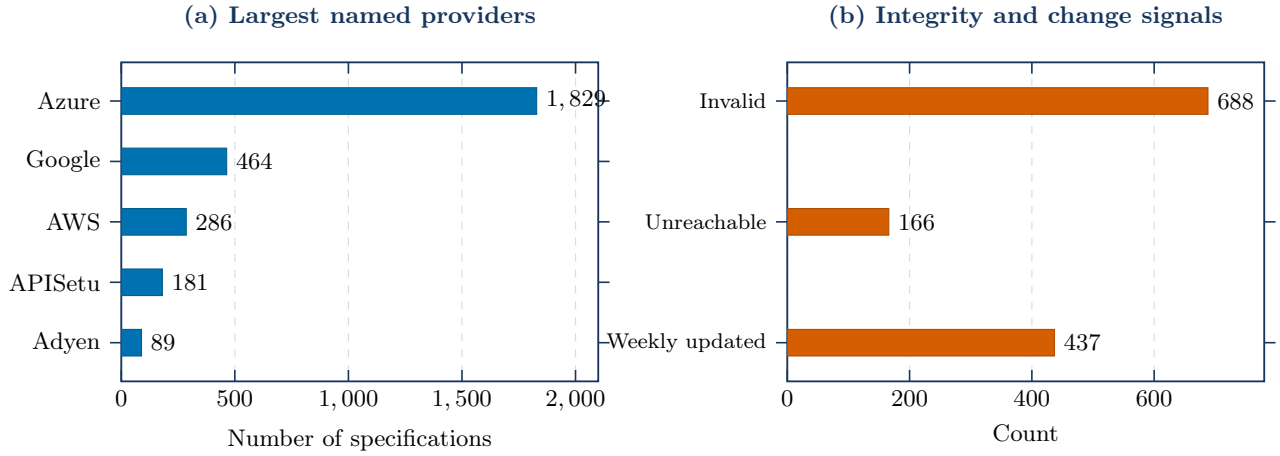

Figure 3: Real public-interface metadata from the APIs.guru metrics endpoint: (a) largest named providers and (b) integrity and change signals.

Table 4: Observable metadata signals in the public OpenAPI snapshot.

Metric	Snapshot value
Specifications	3,992
API descriptions	2,529
Endpoints	108,837
Providers	677
Invalid specifications	688
Unreachable specifications	166
Applied fixes	84,860
Updated in reported week	437

```
import requests

URL = "https://api.apis.guru/v2/metrics.json"
m = requests.get(URL, timeout=30).json()

specs = m["numSpecs"]
providers = m["datasets"][0]["data"]
named = {k: v for k, v in providers.items()
         if k != "Others"}
top4 = sorted(named.items(),
              key=lambda kv: kv[1],
              reverse=True)[:4]

top4_share = sum(v for _, v in top4) / specs
invalid_rate = m["invalid"] / specs
unreachable_rate = m["unreachable"] / specs

print("top4 share:", top4_share)
print("invalid:", invalid_rate)
print("unreachable:", unreachable_rate)
print("weekly updated:", m["thisWeek"]
["updated"])
```

Listing 1 deliberately refreshes the source instead of hard-coding a "real-time" claim into the paper. The dataset packaged with the manuscript records the snapshot used for the figures, while the code makes later updates reproducible.

6. Illustrative Application

To show how the properties discriminate, we apply them to three interface archetypes that recur across enterprise estates. These are composites drawn from documented industry patterns rather than observed cases, and are offered to demonstrate the instrument rather than to evidence it.

6.1. The master data feed

A customer master extract, originally built for one downstream system, now consumed by nine. Structure is declared, the transformation is readable, and the consumer list is maintained because the interface is operationally visible. Coupling breadth is high, specification recoverability is documented, consumer volatility is moderate, and semantic divergence is low because consumers adopted the source definition. This is the *coordination* profile. The remediation is abstraction: the interface is a strong candidate for an intermediating contract, and the economics favour it because volatility guarantees the abstraction will be exercised. Nothing prevents the work except the funding model, which is precisely the condition described as M1.

6.2. The settlement bridge

A nightly reconciliation between a finance system and a partner platform, built fifteen years ago, running unmodified. Two consumers, both stable. The transformation contains conditional logic whose business rationale is recorded nowhere, and the original team has dispersed. Coupling breadth is low, consumer volatility is low, but specification recoverability is opaque and semantic divergence is moderate, since the conditional logic almost certainly encodes a reconciliation rule that neither endpoint states. This is the *knowledge* profile. Its low coupling makes it appear a low priority under any traffic-based or consumer-count-based ranking, which is the error the assessment is designed to prevent. Its remediation cost is not high; it is *unbounded*, because no one can currently state what correct behaviour is. It must be recovered before anything that depends on it is touched.

Table 5: Three archetypes assessed.

Archetype	P1	P2	P3	P4	Constraint
A1 Master feed	High	Doc.	Med	Low	Funding model
A2 Settlement	Low	Opaque	Low	Med	Knowledge loss
A3 Remnant	High	Mixed	High	High	Governance gap

6.3. The acquisition remnant

A set of interfaces inherited through a merger, connecting systems that use a different definition of the same core entity, running on a platform retained for no other purpose. Coupling breadth is high, recoverability is mixed, volatility is high because the acquired estate is being progressively absorbed, and semantic divergence is high by construction. This is the *structural* profile, and it exhibits M2 and M4 simultaneously. The important conclusion is negative: no integration remediation resolves it. Reconciling the definitions in transformation logic creates a third definition owned by no one, which is the mechanism that produced the problem. The prerequisite is a governance decision about which definition is authoritative, and until that decision exists, integration work can only relocate the divergence.

The three cases (6.1 (A1) to 6.3 (A3)) differ in what blocks remediation, and none of the three blockers is technical. A1 is blocked by how work is funded, A2 by what is no longer known, and A3 by who is entitled to decide. An assessment that reported only interface counts or traffic volumes would rank A1 first and A2 last, which inverts the correct order: A2's opacity will halt any programme that later depends on it, while A1's remediation is well understood and merely unfunded.

This is the practical claim of the paper. Integration remediation fails less often for want of engineering than for want of an ordering that reflects where the real constraints lie.

7. Discussion

7.1. Sequencing is the practical contribution

Organizations attempting integration remediation commonly begin with the interfaces that are most visible, which are usually the highest-traffic ones. The properties above suggest a different order. Opaque interfaces should be addressed first, not because they are the most costly to operate but because they are the only ones whose remediation cost is unbounded. Every subsequent programme that touches them inherits that unbounded risk.

7.2. The assessment is deliberately coarse.

Precision here is false economy. An estate with several hundred interfaces cannot be assessed precisely at reasonable cost, and an assessment that is not performed has no value. Three-level scales assessed from existing artifacts are sufficient to establish ordering, which is what sequencing requires.

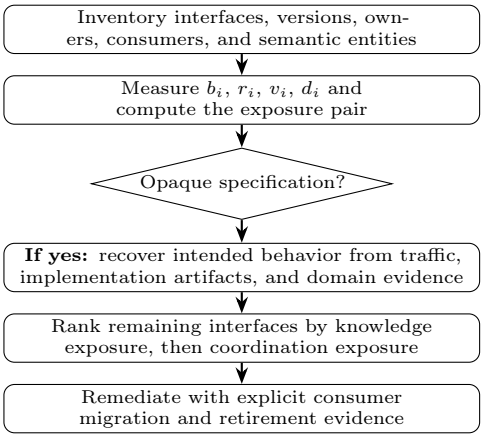

Figure 4: Assessment-to-remediation flow.

7.3. Relationship to platform selection.

The properties bear directly on platform choice. Coupling breadth and consumer volatility jointly determine whether reuse-oriented design is economically justified for a given interface, and semantic divergence determines whether any platform can resolve the underlying problem. A platform decision made without reference to these properties is a decision made on the basis of the interface's construction cost alone, which is the smallest of the costs it will incur.

7.4. What this does not address.

We do not model the financial value of remediation. The technical debt literature has attempted monetary quantification with mixed results [1], [19], and the additional coordination variables in the integration case make a credible monetary model harder rather than easier. We regard ordering as achievable and pricing as not yet achievable.

8. Results

The assessment produces four principal results. They are analytical and illustrative rather than statistically calibrated: the public OpenAPI corpus supplies observable portfolio signals, while the three archetypes test the ordering logic of the proposed framework rather than estimate the prevalence of integration debt in private enterprises.

i) R1: Integration debt cannot be prioritized correctly from interface volume alone

The framework separates coordination exposure from knowledge exposure through (6) and gives opacity lexicographic priority in (7). This distinction changes the order of the illustrative cases. A1 has broad consumption and is a clear abstraction candidate, but its behavior is documented and its blocking condition is funding. A2 has only two stable consumers, yet its specification is opaque; therefore it must be recovered before a modernization programme can safely depend on it. A traffic- or consumer-count-only ranking would place A1 before A2 and would invert the framework's risk-based order.

i) R2: Semantic divergence creates a governance boundary that integration engineering cannot remove

A3 combines high breadth, high volatility, mixed recoverability, and high semantic divergence. The analysis

therefore places it in the structural profile. Its blocking condition is not transformation complexity but the absence of an authoritative business definition. Adding another mapping would encode a third definition and relocate the inconsistency rather than repay the debt. The result distinguishes debt that can be engineered away from debt whose prerequisite is an organizational decision.

i) R3: The public interface snapshot demonstrates that portfolio shape, integrity, and change are independent observables

The APIs.guru snapshot contains 3,992 specifications across 677 providers. The largest named provider contributes 1,829 specifications, or 45.82% of the total, and the four largest named providers together account for 69.14%. At the same time, 688 specifications are reported invalid and 166 unreachable, equal to 17.23% and 4.16% of the specification count, while 437 specifications were updated in the reported week. These measures describe different portfolio conditions: concentration captures dependency shape, invalidity and reachability capture specification integrity, and weekly updates capture change pressure. No single interface count contains all three.

i) R4: The resulting remediation sequence is knowledge-first

Across the mechanisms, properties, and archetypes, the stable ordering is to recover opaque specifications first, then address high knowledge exposure E_i^{know}, and only then optimize coordination exposure E_i^{coord}. This ordering is the main operational result of the paper. It protects modernization programmes from encountering unrecoverable behavior after dependent interfaces have already been changed, while still directing abstraction investment toward interfaces whose breadth and volatility make future coordination cost likely to recur.

9. Limitations

Three limitations bound this work. First, the mechanisms are derived from the literature and from documented industry practice rather than from a systematic empirical study, and we make no claim that they are exhaustive. Second, the four properties are proposed on analytical grounds and have not been calibrated; we do not know their relative weight, nor whether they are independent. Third, the estate profiles of Table 3 are illustrative compositions rather than empirically observed clusters.

Validation would require longitudinal data of a kind rarely collected: interface inventories assessed at two points separated by several years, with recorded remediation attempts, their coordination cost, and their outcome. We regard the collection of such a dataset as the principal obstacle to progress in this area, and a more useful next contribution than further analytical refinement.

10. Conclusion

Integration debt is not architectural technical debt occurring in interfaces. It differs in ownership, in who bears its cost, in when that cost appears, and in whether the indebted artifact carries its own specification. Those differences make the standard remedy of incremental refactoring largely unavailable, which explains why integration estates decay in a characteristic way that the technical debt literature does not predict.

We have characterized four accumulation mechanisms and proposed four observable properties for assessing them. The properties are coarse by design and are intended to establish ordering rather than valuation. We operationalized them as measurable breadth, recoverability, volatility, and semantic-divergence indicators and retained two exposure coordinates rather than forcing an unsupported single debt score. A reproducible public OpenAPI snapshot demonstrates how portfolio concentration, validity, reachability, repair history, and change rate can be collected as distinct metadata signals. The practical use is sequencing: knowledge loss must be addressed before coordination cost, because remediation that encounters unrecoverable specification mid-programme loses the reversibility on which incremental change depends.

Declarations and ethical statements

Conflict of interest: The author declares that there is no conflict of interest.

Funding statement: The author declares that no specific funding was received for this research.

Artificial Intelligence usage statement: During the preparation of this manuscript, the author utilized ChatGPT solely for language refinement and grammatical corrections. The author carefully reviewed and revised the generated content and take full responsibility for the accuracy, integrity, and originality of the final manuscript.

Availability of data and materials: The data and/or materials that support the findings of this study are available from the corresponding author upon reasonable request.

CRediT authorship contribution statement

Anil Chandra Rajendra Pallikonda: Conceptualization, Methodology, Visualization, Writing- Original draft & editing.

Publisher's note

Krrish Scientific Publications Pvt. Ltd. and the ***Journal of Computing and Data Technology*** remain neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

[1] Alves NSR, Mendes TS, de Mendonça MG, Spínola RO, Shull F, Seaman C. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*. 2016;70:100–121. Available from: https://doi.org/10.1016/j.infsof.2015.10.008

[2] Besker T, Martini A, Bosch J. Managing architectural technical debt: A unified model and systematic literature review. *Journal of Systems and Software*. 2018;135:1–16. Available from: https://doi.org/10.1016/j.jss.2017.09.025

[3] Verdecchia R, Malavolta I, Lago P. Architectural technical debt identification: The research landscape. In: *Proceedings of the 2018 International Conference on Technical Debt*. 2018. p. 11–20. Available from: https://doi.org/10.1145/3194164.3194176

[4] Martini A, Bosch J, Chaudron M. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology*. 2015;67:237–253. Available from: https://doi.org/10.1016/j.infsof.2015.07.005

[5] Li Z, Avgeriou P, Liang P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*. 2015;101:193–220. Available from: https://doi.org/10.1016/j.jss.2014.12.027

[6] Tang X, Sun S, Yuan X, Chen D. Automated web service composition system on enterprise service bus. In: *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. 2009. p. 9–13. Available from: https://doi.org/10.1109/SSIRI.2009.24

[7] Galster M, Lapre L, Avgeriou P. SOA in variability-intensive environments: Pitfalls and best practices. *IEEE Software*. 2014;31(1):77–84. Available from: https://doi.org/10.1109/MS.2013.21

[8] Kim JS, Andrade H, Sussman A. Principles for designing data-/compute-intensive distributed applications and middleware systems for heterogeneous environments. *Journal of Parallel and Distributed Computing*. 2007;67(7):755–771. Available from: https://doi.org/10.1016/j.jpdc.2007.04.006

[9] Velepucha V, Flores P. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE Access*. 2023;11:88339–88358. Available from: https://doi.org/10.1109/ACCESS.2023.3305687

[10] Soldani J, Tamburri DA, Van Den Heuvel WJ. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*. 2018;146:215–232. Available from: https://doi.org/10.1016/j.jss.2018.09.082

[11] Kleehaus M, Matthes F. Challenges in documenting microservice-based IT landscape: A survey from an enterprise architecture management perspective. In: *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. 2019. p. 11–20. Available from: https://doi.org/10.1109/EDOC.2019.00012

[12] Alonso J, Orue-Echevarria L, Casola V, Torre AI, Huarte M, Osaba E, Lobo JL. Understanding the challenges and novel architectural models of multi-cloud native applications: A systematic literature review. *Journal of Cloud Computing*. 2023;12:6. Available from: https://doi.org/10.1186/s13677-022-00367-6

[13] Akidau T, Bradshaw R, Chambers C, Chernyak S, Fernández-Moctezuma RJ, Lax R, McVeety S, Mills D, Perry F, Schmidt E, Whittle S. The Dataflow Model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*. 2015;8(12):1792–1803. Available from: https://doi.org/10.14778/2824032.2824076

[14] Overeem M, Mathijssen M, Jansen S. API-m-FAMM: A focus area maturity model for API management. *Information and Software Technology*. 2022;147:106890. Available from: https://doi.org/10.1016/j.infsof.2022.106890

[15] Hyrynsalmi SM. The state-of-the-art of the integration platforms as a service research. In: *2022 IEEE/ACM International Workshop on Software-Intensive Business (IWSiB)*. 2022. p. 17–22. Available from: https://doi.org/10.1145/3524614.3528634

[16] Rahm E, Bernstein PA. A survey of approaches to automatic schema matching. *The VLDB Journal*. 2001;10(4):334–350. Available from: https://doi.org/10.1007/s007780100057

[17] Wang RY, Strong DM. Beyond accuracy: What data quality means to data consumers. *Journal of Management Information Systems*. 1996;12(4):5–33. Available from: https://doi.org/10.1080/07421222.1996.11518099

[18] Kruchten P, Nord RL, Ozkaya I. Technical debt: From metaphor to theory and practice. *IEEE Software*. 2012;29(6):18–21. Available from: https://doi.org/10.1109/MS.2012.167

[19] Ampatzoglou A, Mittas N, Tsintzira AA, Ampatzoglou A, Arvanitou EM, Chatzigeorgiou A, Avgeriou P, Angelis L. Exploring the relation between technical debt principal and interest: An empirical approach. *Information and Software Technology*. 2020;128:106391. Available from: https://doi.org/10.1016/j.infsof.2020.106391

[20] Lenarduzzi V, Besker T, Taibi D, Martini A, Arcelli Fontana F. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software*. 2021;171:110827. Available from: https://doi.org/10.1016/j.jss.2020.110827

[21] Espinha T, Zaidman A, Gross HG. Web API growing pains: Loosely coupled yet strongly tied. *Journal of Systems and Software*. 2015;100:27–43. Available from: https://doi.org/10.1016/j.jss.2014.10.014

[22] Koçi R, Franch X, Jovanovic P, Abelló A. Web API evolution patterns: A usage-driven approach. *Journal of Systems and Software*. 2023;198:111609. Available from: https://doi.org/10.1016/j.jss.2023.111609