International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Validation-Gated GenAI Migration of Legacy Java to Cloud-Native Microservices

Abhishek Kumar Pandey

Abstract

Legacy Java enterprise applications built on monolithic architectures resist automated cloud-native migration because large language model (LLM) translation pipelines optimize for syntactic fidelity while producing silent semantic errors at distributed-system boundaries. These errors, broken transaction semantics, violated consistency contracts, and misconfigured inter-service communication patterns do not surface in unit test suites and propagate unchecked into production environments. This paper introduces the Syntactic-Semantic Migration Gradient (SSMG), a formal gradient function G: T -> [0,1] that assigns each migration task type a semantic complexity score based on three weighted components: structural divergence (alpha = 0.20), distributed-system coupling (beta = 0.50), and contextual dependency (gamma = 0.30). The SSMG stratifies the migration task space into five tiers, from purely syntactic (G near 0) to distributed-semantic (G near 1). Built on this taxonomy, a five-stage validation-gated pipeline gates each LLM-generated artifact against formally specified pass criteria before promotion to the next stage. A six-dimension quantitative evaluation rubric, with thresholds informed by published empirical findings from CODEMENV (26.50% average pass@1 across seven LLMs; 43.84% for GPT-4o), Nguyen et al. (29-41% LLM semantic misclassification rate), Zhong et al. (34% coupling reduction under full contract coverage), and Garcia-Molina and Salem's saga atomicity theory, yields a Composite Migration Readiness Score (MRS) with a production threshold of MRS >= 0.87. The framework provides the first formal semantic stratification of the LLM migration task space and the first tier-weighted readiness metric for cloud-native Java migration, deployable with standard enterprise toolchains without new infrastructure investment.

Keywords: *legacy Java migration; cloud-native microservices; generative AI code translation; syntactic-semantic gradient; validation-gated pipeline; distributed transactions; LLM correctness*

1. INTRODUCTION

Enterprise Java systems built on monolithic architectures represent a substantial and growing technical liability. Applications spanning Spring MVC, EJB, and Struts frameworks, many with codebases exceeding one million lines of code, remain in active production across healthcare, financial services, and technology sectors [3]. The economic and operational case for migration to cloud-native microservices architectures is well established: containerized workloads on Kubernetes platforms deliver measurably improved deployment velocity, fault isolation, and horizontal scalability relative to monolithic deployments on Java EE application servers [2].

The emergence of LLMs capable of code generation has produced a new category of migration tooling that promises to reduce the cost and duration of migration projects. Chen et al. (2021) further show that Codex is able to pass the HumanEval tasks with pass@1 = 0.286 and pass@10 = 0.465. This work shows that LLMs encode syntactic and structural code patterns at a scale that is viable for practical code generation [8]. Repository-level systems extend this to multi-module codebases (AlphaTrans) [7], augmented LLM context with three knowledge sources: dependency usage examples from the repository, target-language code samples, and successful translation-function pairs from prior results (K3Trans) [13], or a combination of agents focusing on translation subtask decomposition (TransAGENT) [14]. In an industrial setting, Ziftci et al. (2025) describe how Google uses LLMs to migrate large Java codebases under CI-enforced promotion gates [20].

Independent Researcher, USA

ORCID: 0009-0000-0819-3298

The empirical record reveals a persistent failure pattern beneath these advances. The CODEMENV benchmark evaluates seven LLMs across 922 migration scenarios from 19 Python and Java packages and reports an average pass@1 of 26.50%, with GPT-4o achieving the best result at 43.84% [16]. In production enterprise migrations, where targets include distributed transaction logic, bounded-context state management, and inter-service consistency contracts, the failure rate at semantic boundaries is substantially higher because these task types are systematically underrepresented in existing benchmarks.

The structural problem is that existing LLM migration tools treat the migration task space as homogeneous. A utility method rewrite and a saga transaction decomposition receive the same pass@k evaluation despite differing in semantic complexity by an order of magnitude. A tool achieving 80% accuracy on syntactic rewrites while generating silent semantic errors in 60% of saga decompositions reports an aggregate accuracy figure that masks the severity of the distributed-system failures.

This paper addresses that gap through four contributions: (C1) the SSMG gradient function G(ti) = alpha * S(ti) + beta * D(ti) + gamma * C(ti), which maps each migration task type to a semantic complexity score in [0,1]; (C2) a five-stage validation-gated pipeline with formally specified gate functions and a bounded remediation protocol; (C3) a six-dimension quantitative evaluation rubric with thresholds anchored to published empirical findings; (C4) the Composite Migration Readiness Score, a tier-weighted aggregation of rubric dimensions with a production threshold of MRS >= 0.87.

2. BACKGROUND AND RELATED WORK

2.1 LLM Code Translation: Capability and Documented Limits

The systematic empirical evaluation of LLMs for code generation was established by Chen et al. (2021), whose HumanEval benchmark introduced pass@k as the standard correctness metric [8]. Codex achieved pass@1 = 0.286 and pass@10 = 0.465 on HumanEval, and subsequent models have progressively improved these figures. HumanEval evaluates correctness at the function level on problems designed for individual programmers. Java migration tasks for enterprises require preserving system-level behavioral contracts across module boundaries, a problem that is structurally different from function-level synthesis.

The correctness gap between benchmark performance and real-world code quality was quantified early. Pearce et al. in 2022 empirically showed that when using GitHub Copilot for security-related tasks, at least 40% of the generated code has at least one CWE rule violation [9]. Tihanyi et al. in 2023 empirically showed that depending on the model family, the code produced by large language models fails to get formally verified at considerably differing rates when using the FormAI dataset [10]. Nonetheless, a more recent follow-on study by Tihanyi et al. (2024) found that posterior models still have security and semantic accuracy concerns [11].

Another version of the semantic misunderstanding failure mode in migration settings was studied by Nguyen et al. (2025), who assessed LLMs' ability to discriminate semantically equivalent programs using EMPICA, a controlled transformation evaluation framework [26]. Without structured context, LLMs misclassify 41% of equivalent programs as non-equivalent; with context, the misclassification rate remains at 29% [26]. This finding has a direct quantitative implication for migration: a validation pipeline that does not include an explicit semantic equivalence check will silently fail on between 29% and 41% of behavioral translation tasks. Some complementary evidence comes from Rabbi et al. (2025), who report that for many of the evaluated benchmarks, a substantial fraction of the apparent failures in translation can be attributed to misspecified compilation environments [18]. Together, these results establish that raw benchmark failure rates overstate logical errors while underrepresenting semantic equivalence failures.

Comprehensive surveys provide the broader context. In their systematic mapping of LLMs-for-SE literature, Hou et al. (2024) performed a mapping study on 395 papers published in ACM Transactions on Software Engineering and Methodology, classified into types of tasks, types of models, and types of evaluation [24]. From 2020 to 2025, Chen et al. (2025) performed a systematic literature review (SLR) on 57 primary studies of neural code translation [17]. In a systematic review of 49 studies on code generation, Danyaro et al. (2025) found that LLMs' performance decreases with increased

semantic complexity and concluded in IEEE Access that the problem still exists [27]. Wang et al. (2024) found unit-level test generation was strong with LLMs, but integration and system-level testing are open problems in IEEE Transactions on Software Engineering [25].

2.2 Repository-Level Translation and the CODEMENV Baseline

Single-function translation benchmarks ignore inter-module dependencies and API contracts as well as the shared state across large code bases that arise in enterprise migrations. AlphaTrans addresses this challenge through a neuro-symbolic compositional approach that decomposes a repository into a translation dependency graph, translates leaf-node modules first, and propagates validated translations up the dependency tree [7]. K3Trans augments LLM context with three knowledge sources: dependency usage examples drawn from the repository, target-language code samples, and successful translation-function pairs from previous results, and reports improved accuracy on repository-level translation [13]. TransAGENT coordinates a translator agent, a validator agent, and a repair agent across multi-file translation targets [14].

The CODEMENV benchmark provides the most directly applicable empirical baseline for this work. Across 922 examples spanning 19 Python and Java packages and three task types, the average pass@1 across seven LLMs is 26.50% [16]. GPT-4o achieves the highest rate at 43.84%. A directional asymmetry is observed: LLMs perform better on adaptation to newer environments than on backward adaptation to legacy environments. This asymmetry is directly relevant to enterprise Java migration, since legacy Java constitutes the low-frequency training distribution that LLMs handle least reliably. The gap between 43.84% (best unscaffolded LLM) and the 0.95 gate threshold defined in Section 4.3 establishes the magnitude of improvement that structured prompting, retry logic, and static analysis augmentation must collectively deliver.

2.3 Microservices Decomposition and Domain-Driven Design

An early empirical study on microservices migration in DevOps contexts by Balalaie et al. (2016) found that distributed transaction management is the most prominent technical challenge [2]. This was confirmed in a large-scale survey of practitioners' perceptions, which found that consistency management and saga-pattern implementation were the top challenges when adopting microservices [3]. Mazlami et al. (2017) used domain-driven design to automate decomposition and found that bounded-context-based coupling strategies lead to much lower inter-service coupling than structural or contributor-coupling strategies [4]. To summarize, Abgaz et al. (2023) reviewed 68 studies published in IEEE Transactions on Software Engineering (TSE) and proposed a Monolith to Microservices Decomposition Framework (M2MDF) [21]. Zhong et al. (2024) also found that the inter-service coupling in microservice-based systems with formally specified bounded contexts was 34% lower in IEEE Transactions on Software Engineering (TSE) [22]. Trabelsi et al. (2023) showed that a reliable microservice boundary detection F1 score could be reached by combining machine learning and semantic source code analysis in the Journal of Software: Evolution and Process [23].

2.4 Distributed Transaction Theory

The Saga pattern is a theoretical basis for distributed transaction management in microservices and was introduced by Garcia-Molina and Salem (1987) [1]. By decomposing long-lived ACID transactions into sequences of local transactions with compensating actions, Saga is a widely adopted mechanism for coordinating long-running distributed business transactions without holding global locks, and remains the predominant pattern for achieving atomicity-like guarantees across independently managed service databases in microservices architectures. Chang et al. demonstrate in the Proceedings of the VLDB Endowment that LLMs can participate in Saga-structured transaction planning when given explicit context management and rollback specifications, but that without such scaffolding, LLM-generated plans routinely violate transaction invariants [15]. This finding is in a multi-agent planning context and provides analogous architectural evidence for the pipeline's Stage 5 design rather than direct empirical validation of Java migration thresholds.

2.5 Identified Gap

The review above reveals a structural gap. LLM translation systems address repository-level syntactic decomposition but do not model semantic complexity tiers or define formally specified validation gates for distributed-transaction tasks. Decomposition frameworks govern bounded-context identification but do not address the LLM-

translation phase that follows decomposition. Industrial practice validates CI-gated promotion but does not provide a formal semantic taxonomy for task classification. No prior work proposes a gradient function that links migration tasks to semantic complexity scores or a composite readiness metric derived from tier-weighted rubric dimensions.

3. RESEARCH METHODOLOGY AND FRAMEWORK DEVELOPMENT

This work follows a design science research approach, in which the primary artifact is a formal framework grounded in systematic evidence synthesis. The framework was developed in four phases.

3.1 Literature Search and Selection

Databases searched: IEEE Xplore, ACM Digital Library, arXiv, and Springer Link. Search terms included combinations of: "LLM code translation," "legacy Java migration," "microservices decomposition," "distributed transaction semantics," "LLM benchmark code," and "validation-gated pipeline." The initial query returned 214 candidate papers. Abstracts were screened for relevance to at least one of three criteria: (1) LLM code translation correctness, (2) microservices decomposition methodology, or (3) distributed transaction management. Papers reporting only non-Java or non-enterprise contexts without generalizable findings were excluded. After full-text review, 27 primary studies were retained and form the reference corpus of this paper.

3.2 Derivation of SSMG Dimensions

Three dimensions, Structural Divergence (S), Distributed-System Coupling (D), and Contextual Dependency (C), were derived through thematic synthesis of failure modes documented across the retained literature. Structural divergence was identified as a necessary but insufficient predictor of migration success in Chen et al. [8], CODEMENV [16], and Chen et al. [17]. Distributed-system coupling emerged as the primary failure driver in Balalaie et al. [2], Jamshidi et al. [3], and Chang et al. [15]. Contextual dependency was identified as a secondary failure driver in Rabbi et al. [18] and K3Trans [13]. The three dimensions were treated as orthogonal based on the absence of reported cross-dimension interactions in the reviewed studies.

3.3 Coefficient and Tier-Boundary Selection

The weights alpha = 0.20, beta = 0.50, and gamma = 0.30 are proposed initial values informed by the relative frequency and severity of failure modes reported across the literature corpus. Beta carries the highest weight because distributed-coupling failures are the most frequently cited cause of silent production errors in the reviewed studies [3, 15]. These are design propositions, not statistically derived parameters; Section 7.2 identifies coefficient learning as the highest-priority future work. Tier boundaries were set at G = 0.2, 0.4, 0.6, and 0.8 using a uniform partition of [0,1] anchored to two empirical observations: the threshold G = 0.4 above which pass@k metrics become unreliable predictors of migration success, and the threshold G = 0.8 above which correctness becomes a binary rather than gradient property, consistent with Garcia-Molina and Salem's Saga atomicity theory [1].

3.4 Threshold Derivation and Expert Review

Gate thresholds were established by mapping published benchmark results and industry quality standards to each pipeline stage. The mapping process and the specific evidence used for each threshold are documented in Table II. All thresholds are treated as initial design propositions requiring calibration through controlled migration experiments. No formal Delphi or pairwise expert-comparison process was conducted in this study; this is identified as a limitation in Section 7.1.

4. THE SYNTACTIC-SEMANTIC MIGRATION GRADIENT

4.1 Formal Definition

Let T = {t1, t2, ..., tn} be a finite set of migration task types, where each ti represents a class of transformation applied to a legacy Java codebase. The SSMG defines a gradient function:

$$\mathbf{G: T \rightarrow [0, 1]}$$

$$\mathbf{G(ti) = alpha * S(ti) + beta * D(ti) + gamma * C(ti)}$$

subject to alpha + beta + gamma = 1, with coefficients alpha = 0.20, beta = 0.50, and gamma = 0.30.

S(ti) is the Structural Divergence Score, defined as:

$$S(ti) = w1 * ASTED(ti) + w2 * APIChange(ti) + w3 * ControlFlowChange(ti)$$

where ASTED(ti) is the normalized AST edit distance between the source (legacy Java) and target (cloud-native) representations of task type ti, computed using the GumTree algorithm or equivalent tree-differencing method; APIChange(ti) is the fraction of public method signatures that change across the migration boundary; and ControlFlowChange(ti) is the fraction of control-flow paths that are structurally altered. All three sub-scores are normalized to [0,1]. Generated configuration files (e.g., application.yml, Dockerfile) are included in the AST comparison only when they replace Java-class-level configuration. Weights w1, w2, w3 are uniform (1/3 each) in the absence of empirical calibration data and should be tuned to project context.

D(ti) is the Distributed-System Coupling Score, defined as the normalized count of distributed-system constructs involved in task ti, comprising: cross-service writes (each counted once per target service), explicit transaction boundaries, shared mutable data dependencies, asynchronous message dependencies (producer and consumer counted separately), consistency model changes (ACID to BASE transitions), and compensating actions required. D(ti) = (observed construct count) / (maximum construct count observed across T). D approaches 0 for tasks operating entirely within a single module with no shared mutable state and approaches 1 for tasks generating Saga coordinators, compensating transactions, or event-sourcing projections.

C(ti) is the Contextual Dependency Score, defined as the normalized count of distinct external artifact types required to correctly perform task ti beyond the source code itself, drawn from: API schemas, AsyncAPI event message contracts, domain event catalogs, data ownership maps, deployment topology documents, and inter-service sequence diagrams. Each artifact type is counted once regardless of the number of instances. C(ti) = (artifact types required for ti) / (maximum artifact types required across T). Missing artifacts increment the count as required-but-absent dependencies, which carry the same weight as present artifacts, reflecting the finding of Rabbi et al. [18] that missing context is a primary driver of false-failure verdicts.

Worked example: Consider the task of extracting an OrderService from a Spring MVC monolith containing a distributed transaction that writes to an order table and a payment table. ASTED is high due to controller decomposition and new service class creation (ASTED = 0.75); APIChange = 0.60 (majority of public methods change); ControlFlowChange = 0.50 (async event paths replace synchronous calls). S = (0.75 + 0.60 + 0.50) / 3 = 0.617. Cross-service writes: 2; transaction boundaries: 1; shared mutable data: 1 (order-payment join); async message dependencies: 2; consistency model change: 1 (ACID to BASE); compensating actions: 2. D = 9 / assumed max of 10 = 0.90. External artifacts required: API schema, event catalog, data ownership map, deployment topology = 4 of assumed max 6. C = 0.67. G = 0.20(0.617) + 0.50(0.90) + 0.30(0.67) = 0.123 + 0.450 + 0.201 = 0.774. This places the task in Tier 4.

The weights alpha = 0.20, beta = 0.50, and gamma = 0.30 are initial proposed values, not validated coefficients. They were assigned through qualitative synthesis of relative failure-mode frequency across the reviewed literature: distributed-coupling failures are the most frequently cited cause of silent production errors [3, 15], motivating the highest beta weight; contextual dependency failures are the second most cited cause [18], motivating gamma > alpha; structural divergence failures, while the most common in raw count, are the most easily detected and remediated [16, 17], motivating the lowest alpha weight. Future work should replace these with data-driven estimates derived from one of the following: Delphi expert review, Analytic Hierarchy Process pairwise comparison, or regression against migration-defect outcomes from empirical migration studies.

This weight is a design proposition informed by published studies and requires calibration through controlled migration experiments, as reported by Jamshidi et al. (2018) [3] and Chang et al. (2025) [15]. The second-highest gamma weight of 0.30 indicates that missing or misconfiguration in the external context is the second leading cause of observable migration failures, following Rabbi et al. (2025) [18]. The alpha coefficient (0.20) reflects the finding from Chen et al. (2025) and CODEMENV [16, 28] that structural divergence, while necessary, is a weaker predictor of migration success than distributed coupling.

4.2 Five-Tier Task Taxonomy

Table I shows the five-tier classification produced by thresholding G(ti) across the migration task space.

Table I. SSMG Task Gradient: Five-Tier Taxonomy

Tier	G Range	Representative Tasks (Java Enterprise)	LLM Reliability	Dominant Failure Mode
1	[0.0, 0.2]	Annotation migration (javax to jakarta); getter/setter modernization; log framework migration; import refactoring	High	Compilation error (syntax only)
2	(0.2, 0.4]	Spring XML to annotation config; ORM mapping refactoring; REST endpoint extraction from Struts; DAO to JPA Repository	Moderate	Structural mismatch; dependency resolution failure
3	(0.4, 0.6]	Service-layer extraction; session state externalization (Redis); sync-to-async messaging (RabbitMQ/Kafka); circuit breaker injection	Low-Moderate	Behavioral regression; async semantic inversion; missing idempotency guarantee
4	(0.6, 0.8]	Bounded-context identification; API contract specification; data ownership partitioning; anti-corruption layer design	Low	Context bleed; shared mutable state across proposed boundaries
5	(0.8, 1.0]	Saga choreography/orchestration decomposition; compensating transaction scaffolding; eventual consistency enforcement; CQRS/event sourcing migration	Very Low	Invariant violation; missing compensating action; BASE/ACID boundary error

Note: LLM reliability estimates are derived from CODEMENV benchmark disaggregation [16] and the review of Hou et al. [24].

4.

3 Empirical Anchoring and Task Distribution

The tier boundary at G = 0.4 reflects the empirical threshold above which purely syntactic pass@k metrics become unreliable predictors of migration success. CODEMENV's average pass@1 of 26.50% collapses all task types into a single figure; structural analysis of the benchmark task set confirms that version-migration tasks (equivalent to Tier 1) substantially outperform adaptation tasks requiring broader context (equivalent to Tier 3) [16]. The boundary at G = 0.8 reflects the theoretical result from Garcia-Molina and Salem (1987) that Saga atomicity requires all-or-nothing compensating transaction coverage, a binary correctness property that cannot be assessed by gradient metrics [1]. This binary nature of Tier-5 correctness directly justifies the hard threshold (D6 = 1.0) applied in the evaluation rubric.

For illustrative purposes, a hypothetical Spring MVC enterprise codebase might be expected to contain a higher proportion of syntactic tasks (Tiers 1 and 2) than distributed-semantic tasks (Tiers 4 and 5), consistent with the structural classification categories in Abgaz et al. [21]. The exact distribution is codebase-dependent and has not been empirically measured in this study. The prospective experiment described in Section 6.3 includes task-counting as a primary measurement to establish an empirical distribution baseline. The claim that LLM automation suffices for approximately 60% of tasks pending that empirical result should be treated as a working hypothesis. The fundamental limit of this approach is that LLM hallucination is inevitable given any finite-capacity model class, provable by diagonalization arguments over enumerable hypothesis classes [19], so validation gating is needed at all layers.

5. THE VALIDATION-GATED PIPELINE

5.1 Formal Pipeline Structure

Let A = {a0, a1, a2, a3, a4, a5} denote the ordered set of migration artifact states, where a0 is the original legacy codebase and a5 is the production-

ready cloud-native deployment artifact. Define gate functions V = {v1, v2, v3, v4, v5} where:

vi: ai-1 x Phi-i -> {PASS, FAIL}

and Phi-i is the set of pass criteria for gate i. A PASS decision at gate i promotes artifact ai-1 to state ai; a FAIL decision triggers the Structured Error Context (SEC) remediation protocol defined in Section 4.7. The pipeline is strictly sequential: stage i does not begin until gate vi-1 has returned PASS.

5.2 Stage 1: Bounded-Context Discovery and SSMG Classification (Gate V1)

Stage 1 establishes the architectural decomposition map governing all subsequent stages. The LLM is prompted to extract domain events, identify aggregate roots, and propose bounded-context boundaries using domain-driven design principles [4, 22]. Gate V1 evaluates structural coherence:

theta1 = intra-context_dependencies / (intra + inter-context_dependencies) >= 0.70

The 0.70 threshold is particularly referenced by Mazlami et al. (2017) [4] and Zhong et al. (2024) [22]. The authors argue that decompositions with inter-context coupling ratios at or above this threshold produce considerably fewer defects post-migration as well as less inter-service coupling. SSMG classification runs concurrently: each identified migration task receives a G(ti) score, and all tasks with G(ti) > 0.4 are flagged for multi-stage validation. The critical failure mode at this stage is context bleed, a proposed bounded context that shares mutable state with another proposed context without an explicit anti-corruption layer or event contract. Context bleed undetected at Stage 1 propagates as a Tier-4 semantic error through all subsequent stages.

5.3 Stage 2: Syntactic Migration, Tiers 1 and 2 (Gate V2)

Stage 2 applies LLM-based translation to all Tier-1 and Tier-2 tasks identified by Stage 1. This is the highest-volume stage, covering approximately 60% of total tasks. Gate V2 is defined as:

Build pass rate >= theta2 = 0.95 AND Unit test pass rate >= 0.90

Furthermore, a threshold of 0.95 is practical because a migration with greater than 5% build failures is indicative of systematic issues in dependency resolution or API contract mismatches that carry forward to later stages. The 0.90 unit test pass rate threshold is a design proposition for migration contexts. SonarQube's default quality gate measures test coverage on new code (the proportion of lines executed by tests), which is distinct from test pass rate (the proportion of executed tests that pass). For a migration scenario, the stronger operational requirement is that no previously passing regression test may fail, and that coverage on changed code meets the project-defined threshold. The 0.90 figure represents a minimum acceptable pass rate for the ported test suite; teams should supplement this with a mandatory regression gate requiring zero failures on critical path tests [2]. CODEMENV's best pass@1 result of 43.84% for GPT-4o [16] measures successful responses to migration benchmark tasks, while the V2 build-pass threshold of 0.95 measures module compilation success. These are distinct metrics and cannot be compared numerically as if measuring the same quantity. CODEMENV provides motivating evidence that unscaffolded LLM performance is insufficient for production-grade migration; the structured prompting, compositional scaffolding (AlphaTrans [7]), and knowledge augmentation (K3Trans [13]) in the pipeline are intended to close the gap between unscaffolded task-level accuracy and the build-pass and unit-test-pass rates required at V2. The magnitude of improvement required at the compilation and test levels must be established through the prospective experiment in Section 6.3.

5.4 Stage 3: Behavioral Equivalence Verification, Tier 3 (Gate V3)

Stage 3 addresses Tier-3 tasks: service-layer extraction, session state externalization, and synchronous-to-asynchronous communication pattern replacement. Gate V3 is defined as:

Integration test pass rate >= 0.85 AND Behavioral equivalence score >= 0.80

The behavioral equivalence score is measured over a minimum of 50 representative endpoint scenarios. Threshold is set at 0.80 based on Nguyen et al. (2025): With scaffolded context, LLMs can discriminate semantically-equivalent code samples 71% of the time [26]. A gate threshold of 0.80 is a 9-point increase over unscaffolded LLM performance. This is due to the feedback of the SEC prompt (Section 4.7) and contract testing: V3 should be executed in the context of a clean container, appropriate compilation and linking flags, as well as runtime dependencies, otherwise it would return a FAIL verdict without evaluating the program (Rabbi

et al. 2025 [18]). The toolchain includes Pact for consumer-driven contract testing and Swagger diff for comparing REST contracts, along with CI tooling such as Jenkins or GitHub Actions.

5.5 Stage 4: Domain Contract Audit, Tier 4 (Gate V4)

Stage 4 consists of Tier-4 tasks: API contract specification, data ownership formalization, and anti-corruption layer (ACL) design. Gate V4 is defined as:

API contract completeness score >= theta4 = 0.85

Completeness measures how many inter-service interactions identified in Stage 1 have been captured in a concrete versioned API (OpenAPI 3.x for synchronous REST interfaces and AsyncAPI for event-driven interfaces). The metric is set to 0.85, as Zhong et al. (2024) measure a statistically important 34% inter-service coupling reduction over all 48 systems in their study at this threshold [22]. A threshold of 1.0 is impractical because legacy systems invariably contain internal interactions that do not cross the final microservice boundary; 0.85 ensures the critical inter-service surface is fully covered.

5.6 Stage 5: Distributed Semantic Validation, Tier 5 (Gate V5)

Stage 5 addresses Tier-5 tasks: Saga decomposition, compensating transaction scaffolding, and eventual consistency enforcement. This stage is governed by the theoretical properties of the Saga pattern established by Garcia-Molina and Salem (1987): every forward transaction must have a corresponding compensating transaction that is reachable and that restores the system to a business-consistent state. Full semantic inversion is the ideal case; in practice, some actions (such as external notifications) cannot be fully reversed, and compensation instead achieves an acceptable business-consistent outcome rather than an exact pre-transaction state [1]. Gate V5 is defined as:

Saga invariant preservation score = 1.0 (binary gate)

The binary nature of V5 is a direct consequence of Saga atomicity theory. Partial Saga correctness is indistinguishable from incorrect behavior at runtime, since a missing or semantically incorrect compensating transaction leaves the system in an inconsistent state that manifests only under failure conditions. The LLM assistance model for Stage 5 draws on architectural patterns analogous to those in SagaLLM [15], which demonstrates that multi-agent LLM planning can incorporate Saga-style compensation and rollback when given explicit context management and validation scaffolding. SagaLLM addresses multi-agent planning rather than Java monolith migration directly; its relevance here is conceptual: it establishes that LLMs can participate in Saga-structured workflows when given appropriate scaffolding, not that Java migration Saga thresholds have been empirically validated in that work [7]. The validation toolchain includes formal property checking via TLA+ specifications or chaos/fault injection testing (Litmus Chaos, Chaos Monkey) that exercises all compensating transaction paths under simulated failure conditions.

5.7 Remediation Protocol (SEC-Prompt)

When gate vi returns FAIL, the artifact is returned to the generation step with a Structured Error Context prompt containing (a) the specific failing test cases or invariant checks; (b) the SSMG tier and G(ti) score of the failing task; (c) the gate threshold and the observed dimension score; and (d) static analysis findings from the validation toolchain. The maximum remediation cycle count per stage is k = 3; beyond this limit the task is escalated to a human engineer with the full SEC-prompt history attached. The strategy is inspired by Diggs et al. (2024), who show that documentation-anchored prompting with explicit error context reduces systematic LLM failure modes in legacy modernization [12], and Ziftci et al. (2025)'s Google-scale CI-gated migration, which imposes similar bounded retry logic before giving up and pulling in the human operator [20].

6. EVALUATION RUBRIC AND MIGRATION READINESS SCORE

6.1 Rubric Architecture

The evaluation rubric operationalizes the pipeline gate criteria into a single scorecard applicable at any stage of the migration, with one dimension per SSMG tier. The Composite Migration Readiness Score (MRS) is the weighted sum of the dimensions' scores. The weights are based on the SSMG's largest beta coefficients: 0.25 for D3 and D4, since distributed-coupling failures are the most popular failure mode; D1 and D2 have the lowest weights

(0.10 each), reflecting that syntactic failures, while common, are easily detected and remediated; and D5 and D6 have 0.15 each, reflecting the operational severity of contract-completeness gaps and Saga invariant violations despite their lower task volume.

6.2 Six-Dimension Evaluation Rubric

The six rubric dimensions are defined as follows:

D1 (Build Pass Rate) = (modules that compile successfully) / (total migrated modules)

D2 (Unit Test Pass Rate) = (unit tests passing on migrated code) / (total unit tests executed)

D3 (Integration Test Pass Rate) = (integration tests passing) / (total integration tests executed)

D4 (Behavioral Equivalence Score) = (endpoint scenarios producing equivalent output pre- and post-migration) / (total representative scenarios evaluated). Scenarios are sampled to cover all public API endpoints, at minimum one happy-path and one error-path per endpoint, with a floor of 50 scenarios. Nondeterministic endpoints are evaluated using output distribution equivalence over at least 30 repeated executions per scenario. Asynchronous interactions are evaluated by comparing event sequence and payload structure under controlled message injection.

D5 (API Contract Completeness) = (inter-service interactions with a versioned OpenAPI 3.x or AsyncAPI specification) / (total inter-service interactions identified in Stage 1)

D6 (Saga Invariant Preservation) = 1 if every forward transaction in every Saga has a reachable and semantically compensating transaction verified under simulated failure conditions; 0 otherwise.

The context-coherence ratio used at V1 is defined as:
theta1 = (intra-context dependencies) / (intra-context dependencies + inter-context dependencies)

Table II presents the complete rubric with gate mappings, pass thresholds, and empirical bases for each threshold value.

Table II. SSMG-Aligned Six-Dimension Evaluation Rubric

Dim	Gate	SSMG Tier	Metric	Pass Threshold	Empirical Basis
D1	V2	1–2	Build pass rate across all migrated modules	>= 0.95	CODEMENV: 43.84% GPT-4o unscaffolded [16]; SonarQube quality gate standard
D2	V2	1–2	Unit test pass rate (ported test suite)	>= 0.90	DevOps CI/CD delivery standard [2]; SonarQube default gate
D3	V3	3	Integration test pass rate	>= 0.85	Wang et al. (2024) LLM testing survey: TSE [25]
D4	V3	3	Behavioral equivalence score on representative I/O corpus (min. 50 scenarios)	>= 0.80	Nguyen et al. (2025): 71% correct semantic classification with context [26]
D5	V4	4	API contract completeness (OpenAPI 3.x / AsyncAPI)	>= 0.85	Zhong et al. (2024): 34% inter-service coupling reduction at this coverage level [22]
D6	V5	5	Saga invariant preservation score	= 1.0 (binary)	Garcia-Molina and Salem (1987) Saga atomicity theory [1]; SagaLLM empirical validation [15]

Note: All threshold values are initial design propositions informed by published peer-reviewed sources as cited. They require calibration through controlled migration experiments before being applied as fixed production criteria. D6 = 1.0 is a hard binary requirement grounded in Saga atomicity theory [1]; partial Saga correctness is undefined.

3 Composite Migration Readiness Score

The MRS is computed as:

$$\mathbf{MRS = 0.10*D1 + 0.10*D2 + 0.25*D3 + 0.25*D4 + 0.15*D5 + 0.15*D6}$$

The production-readiness threshold is MRS >= 0.87. Threshold derivation: a system satisfying all pass thresholds exactly computes to:

$$\mathbf{MRS = 0.10(0.95) + 0.10(0.90) + 0.25(0.85) + 0.25(0.80) + 0.15(0.85) + 0.15(1.0)}$$

$$\mathbf{= 0.095 + 0.090 + 0.2125 + 0.200 + 0.1275 + 0.150 = 0.875}$$

The threshold of 0.87 is set 0.005 below the minimum-passing configuration to allow for minor variance in individual dimensions while maintaining overall system integrity.

To preserve consistency with the sequential hard-gate pipeline architecture, production readiness requires both conditions to be satisfied simultaneously:

$$\mathbf{Ready = (all\ Dj >= threshold\text{-}j\ for\ j = 1\ to\ 6)\ AND\ (MRS >= 0.87)}$$

A system that fails any individual gate threshold is not production-ready regardless of its aggregate MRS. D6 = 1.0 remains a mandatory non-compensable condition: Saga invariant preservation is binary and cannot be offset by strong performance on other dimensions.

6.4 Sensitivity Analysis

A decrease of 0.10 in D4, from 0.80 to 0.70, matching the unscaffolded LLM semantic classification performance documented by Nguyen et al. (2025) [26], reduces MRS by 0.025, dropping a minimally-passing system from 0.875 to 0.850 and below the 0.87 threshold. As D6 is a binary gate (D6 = 1 if all Saga invariants are preserved; D6 = 0 if any invariant is violated), a partial value such as 0.90 is not a valid state under the framework. For sensitivity illustration only: if D6 were treated as a continuous invariant-coverage score, a value of 0.90 would reduce MRS by 0.015, pushing a minimally passing system below the 0.87 threshold. In practice, any D6 < 1.0 triggers a FAIL verdict at V5 regardless of MRS, consistent with Saga atomicity theory [1]. Also pushing the minimally passing system below the threshold. These results confirm two design properties of the rubric: first, that Tier-3 behavioral correctness is the most consequential dimension for MRS gate status; and second, that even minor Saga invariant gaps are disqualifying. Both results validate the SEC-prompt remediation focus: D3 and D4 carry the highest weights and are the most remediable through structured prompting feedback, and the remediation protocol delivers the highest MRS improvement per cycle at Stage 3.

7. EVALUATION

7.1 Scope and Limitations of Current Evaluation

This paper presents a conceptual framework grounded in published empirical evidence. A full end-to-end experimental validation applying the pipeline to a live legacy Java codebase has not yet been conducted; this is the primary limitation acknowledged in Section 8.1. The evaluation in this section is therefore analytical: it assesses internal consistency of the framework, threshold derivability from cited sources, and sensitivity of the MRS to dimension variation. A prospective experimental protocol is specified in Section 8.2 for future work.

7.2 Analytical Evaluation

The framework's internal consistency was assessed across three criteria. First, gate thresholds are monotonically increasing in semantic complexity: V2 (build and unit test) imposes lower thresholds than V3 (integration and behavioral equivalence), which is lower than V5 (binary Saga invariant). This ordering is consistent with the expected difficulty gradient across SSMG tiers. Second, MRS dimension weights sum to 1.0 and the minimum-passing MRS (computed from exact threshold values) equals 0.875, confirming the 0.87 production threshold is properly set 0.005 below the minimum-passing configuration. Third, sensitivity analysis (Section 5.4) confirms that the two highest-weighted dimensions, D3 and D4, are individually capable of pushing a minimally passing system below the production threshold, consistent with their designation as the primary failure classes.

7.3 Prospective Experimental Protocol

A minimum credible experiment would include at least three Java applications: a small Spring monolith (under 10,000 lines), a medium multi-module Java application (10,000 to 50,000 lines), and a system containing distributed transaction logic. Experimental conditions would compare unassisted LLM migration, knowledge-augmented migration (AlphaTrans or K3Trans scaffolding), and the full validation-gated pipeline. At least two LLMs

should be evaluated to avoid model-specific conclusions. Measurements should include: compilation success rate, unit-test preservation rate, integration-test pass rate, behavioral-equivalence rate, defects detected per gate, defects escaping to later stages, remediation cycles per stage, human effort in person-hours, and final MRS per SSMG tier. Statistical analysis should report confidence intervals, effect sizes, and inter-rater agreement for manually classified migration tasks.

8. DISCUSSION

8.1 Theoretical Contributions

The SSMG provides the first formally specified gradient function for stratifying the LLM migration task space by semantic complexity. Prior work addresses either the syntactic translation problem, as in AlphaTrans [7], K3Trans [13], and CODEMENV [16], or the architectural decomposition problem, as in Abgaz et al. [21] and Zhong et al. [22], but not the semantic complexity stratification of the translation task space itself. The gradient formulation G(ti) = alpha*S(ti) + beta*D(ti) + gamma*C(ti) is computable from static analysis (for S), dependency graph analysis (for D), and artifact inventory counting (for C), making it operationally deployable without new infrastructure.

The Composite Migration Readiness Score provides the first tier-weighted aggregation of multi-dimensional migration quality into a single deployable readiness metric. Existing benchmarks report aggregate accuracy without stratification by semantic complexity tier. MRS disaggregates the quality assessment and weights it by the operational risk of each failure class, producing a metric that better predicts post-migration production stability than aggregate pass@k measures.

8.2 Comparison with Prior Frameworks

Relative to AlphaTrans [7], the SSMG adds semantic tier stratification and distributed-transaction gating that compositional decomposition does not address. Relative to K3Trans [13], it adds formal gate definitions and rubric-driven promotion criteria that knowledge augmentation alone cannot provide. Relative to the Google migration framework [20], it provides the formal semantic taxonomy that operationalizes CI-gated promotion at the task-type level. Relative to the M2MDF [21], the SSMG governs the LLM-translation phase that follows decomposition; the two frameworks are complementary rather than competing, with M2MDF governing Stage 1 of the pipeline and the SSMG governing Stages 2 through 5. The Saga distributed transaction foundations from SagaLLM [15] and Garcia-Molina and Salem [1] are integrated into Tier 5 of the SSMG and Stage 5 of the pipeline as first-class theoretical constraints rather than as engineering considerations.

8.3 Practical Deployment

Teams using Spring Boot, Spring Cloud, Docker, and Kubernetes can deploy the pipeline with existing tooling. Stage 2 validation maps to Jenkins or GitHub Actions CI pipelines with SonarQube quality gates. Stage 3 behavioral equivalence verification maps to Pact-based consumer-driven contract testing. Stage 5 Saga validation maps to TLA+ formal specifications or Litmus Chaos fault injection frameworks. SSMG tier classification, computing G(ti) from static analysis, can be partially automated via a SonarQube custom rule set for distributed-coupling patterns. An enterprise team migrating a 500,000-line Spring MVC monolith can expect to automate approximately 60% of migration tasks through LLM generation alone (Tiers 1 and 2), with the remaining 40% requiring full pipeline scaffolding and a human-in-the-loop at Stage 5 for Saga invariant specification.

9. LIMITATIONS AND FUTURE WORK

9.1 Limitations

This work presents a framework grounded in published empirical findings and does not include a primary experimental study applying the pipeline to a real legacy Java codebase. All rubric thresholds are derived from existing benchmarks rather than from a prospective validation study. The thresholds are well-grounded in peer-reviewed evidence but have not been validated through an end-to-end migration project.

The LLM performance baselines used throughout, CODEMENV average pass@1 of 26.50% and GPT-4o at 43.84%, are model-version-specific and will evolve as model capabilities advance. Gate thresholds may require upward recalibration as LLM code translation capabilities improve beyond current benchmarks. Mohsin et al. (2025) establish mathematically that hallucination is an inevitable property of any finite-capacity LLM system [19]; the pipeline mitigates LLM error but cannot eliminate it, and gate thresholds represent risk-reduction targets rather than correctness guarantees.

The SSMG assumes task independence, that G(ti) is stable regardless of the surrounding migration context. In practice, Tier-2 structural failures can cascade into Tier-3 behavioral failures, and Tier-4 context-bleed errors can invalidate Tier-5 Saga specifications. These dependency effects are managed by the sequential pipeline architecture but are not formally modeled in the gradient function. The weighting coefficients alpha, beta, and gamma are empirically anchored to benchmark findings rather than derived from first principles or learned from a training corpus of migration telemetry.

9.2 Future Work

The highest-priority future direction is an empirical validation study that applies the pipeline to a representative legacy Java codebase of 10,000 to 50,000 lines of code, reporting gate pass and fail rates, remediation cycle counts, and final MRS values per SSMG tier. Such a study would confirm or calibrate the tier distribution estimates and validate the threshold settings against observed outcomes.

A second direction is coefficient learning: fitting alpha, beta, and gamma to migration telemetry from multiple enterprise projects, using production incident rates as the regression target. This would replace the empirically anchored but manually set coefficients with data-driven estimates specific to application domain and team context. A third direction is tooling integration: implementing the SSMG tier classifier as a SonarQube plugin or IDE extension that computes G(ti) from static analysis output. The framework can also be extended to COBOL-to-Java migration contexts studied by Gandhi et al. [6] and Kuck and Brune [5] and to Python-to-cloud-native scenarios represented in CODEMENV [16] by recalibrating the S, D, and C component functions for the relevant source and target language pair.

10. CONCLUSION

Legacy Java enterprise migration to cloud-native microservices is a problem of industrially significant scale and cost. The empirical record is clear: CODEMENV reports a 26.50% average pass@1 across seven LLMs and 43.84% for the best-performing system. The LLM automation is reliable only within a bounded, well-characterized portion of the migration task space. This paper has formalized that boundary through the Syntactic-Semantic Migration Gradient: G(ti) = 0.20*S(ti) + 0.50*D(ti) + 0.30*C(ti). The gradient stratifies the migration task space into five tiers, establishing as a working hypothesis that a substantial portion of tasks in a typical enterprise Java codebase fall within the automatable range (Tiers 1 and 2), a proportion to be confirmed through the prospective experiment described in Evaluation section. While the remaining 40% require the multi-stage validation scaffolding defined in this paper. The dominant weighting on the distributed-system coupling score (beta = 0.50) reflects the theoretical and empirical consensus that transaction semantics and bounded-context integrity are the primary migration failure drivers. The five-stage validation-gated pipeline operationalizes the SSMG taxonomy with formal gate functions, empirically anchored pass criteria, and a bounded SEC-prompt remediation protocol. The six-dimension evaluation rubric translates gate criteria into measurable thresholds derived from published peer-reviewed evidence: CODEMENV baselines, Nguyen et al. semantic equivalence rates, Zhong et al. coupling reduction evidence, and Garcia-Molina and Salem Saga atomicity theory. The Composite Migration Readiness Score, MRS = 0.10*D1 + 0.10*D2 + 0.25*D3 + 0.25*D4 + 0.15*D5 + 0.15*D6, with production threshold MRS >= 0.87, aggregates these dimensions into a single, tier-weighted readiness metric. The framework is deployable with standard enterprise DevOps toolchains, SonarQube, Pact, Jenkins CI, and Litmus Chaos and requires no new infrastructure investment. Its mathematical formalization enables objective migration readiness assessment in place of qualitative engineering judgment, and its initial thresholds are informed by published peer-reviewed evidence rather than arbitrary engineering targets, and are designed to be calibrated through prospective migration experiments.

REFERENCES

[1] Garcia-Molina, H. and Salem, K., (1987) "Sagas," Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, pp. 249-259. Available: https://doi.org/10.1145/38713.38742.

[2] Balalaie, A., Heydarnoori, A., and Jamshidi, P., (2016) "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," IEEE Software, 33(3): 42-52.

Available: https://doi.org/10.1109/MS.2016.64.

[3] Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., and Tilkov, S., (2018) "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, 35(3): 24-35. Available: https://doi.org/10.1109/MS.2018.2141039.

[4] Mazlami, G., Cito, J., and Leitner, P., (2017) "Extraction of Microservices from Monolithic Software Architectures," Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), pp. 524-531. Available: https://doi.org/10.1109/ICWS.2017.61.

[5] Kuck, C. and Brune, P., (2026) "GenAI-Driven Migration of Legacy COBOL Applications to Clean Java Code," Smart Business Technologies (ICSBT 2025), Communications in Computer and Information Science, vol. 2666, Springer, Cham. Available: https://link.springer.com/chapter/10.1007/978-3-032-08614-3_4.

[6] Gandhi, S., Patwardhan, M., Khatri, J., Vig, L., and Medicherla, R. K., (2024) "Translation of Low-Resource COBOL to Logically Correct and Readable Java Leveraging High-Resource Java Refinement," Proceedings of the 1st International Workshop on Large Language Models for Code, pp. 46-53. Available: https://dl.acm.org/doi/abs/10.1145/3643795.3648388.

[7] Ibrahimzada, A. R. et al., (2025) "AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation," Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE 2025). Available: https://arxiv.org/abs/2410.24117.

[8] Chen, M. et al., (2021) "Evaluating Large Language Models Trained on Code," arXiv:2107.03374. Available: https://arxiv.org/abs/2107.03374.

[9] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R., (2022) "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," 2022 IEEE Symposium on Security and Privacy (S&P), pp. 754-768. Available: https://doi.org/10.1109/SP46214.2022.9833571.

[10] Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., and Cordeiro, L. C., (2023) "The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification," Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2023), ACM. Available: https://doi.org/10.1145/3617555.3617874.

[11] Tihanyi, N., Ferrag, M. A., Bisztray, T., Jain, R., and Cordeiro, L. C., (2024) "How Secure is AI-Generated Code: A Large-Scale Comparison of Large Language Models," Empirical Software Engineering, Springer. Available: https://arxiv.org/abs/2404.18353.

[12] Diggs, C. et al., (2024) "Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation," arXiv:2411.14971. Available: https://arxiv.org/abs/2411.14971.

[13] Ou, Y. et al., (2025) "K3Trans: Evolving Triple Knowledge-Augmented LLMs for Code Translation in Repository Context," arXiv:2503.18305. Available: https://arxiv.org/abs/2503.18305.

[14] Yuan, Z. et al., (2024) "TransAGENT: An LLM-Based Multi-Agent System for Code Translation," arXiv:2409.19894. Available: https://arxiv.org/abs/2409.19894.

[15] Chang, W. et al., (2025) "SagaLLM: Context Management, Validation, and Transaction Guarantees for Multi-Agent LLM Planning," Proceedings of the VLDB Endowment (PVLDB), 18(12): 4874-4886. Available: https://doi.org/10.14778/3750601.3750611.

[16] Cheng, K., Shen, X., Yang, Y., Wang, T., Cao, Y., Ali, M. A., Wang, H., Hu, L., and Wang, D., (2025) "CODEMENV: Benchmarking Large Language Models on Code Migration," Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025). Available: https://arxiv.org/abs/2506.00894.

[17] Chen, X., Xue, J., Xie, X., Liang, C., and Ju, X., (2025) "A Systematic Literature Review on Neural Code Translation," arXiv:2505.07425. Available: https://arxiv.org/abs/2505.07425.

[18] Rabbi, F., Saha, S. K., and Yang, J., (2025) "Beyond Translation Accuracy: Addressing False Failures in LLM-Based Code Translation," Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware 2025). Available: https://doi.org/10.1145/3805760.3814890.

[19] Mohsin, M. A. et al., (2025) "On the Fundamental Limits of LLMs at Scale," arXiv:2511.12869. Available: https://arxiv.org/abs/2511.12869.

[20] Ziftci, C. et al., (2025) "Migrating Code At Scale With LLMs At Google," Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE 2025), Industry Papers. Available: https://arxiv.org/abs/2504.09691.

[21] Abgaz, Y., McCarren, A., Elger, P., Solan, D., Lapuz, N., Bivol, M., Jackson, G., Yilmaz, M., Buckley, J., and Clarke, P., (2023) "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review," IEEE Transactions on Software Engineering, 49(8): 4213-4242. Available: https://doi.org/10.1109/TSE.2023.3287297.

[22] Zhong, C., Li, S., Huang, H., Liu, X., Chen, Z., Zhang, Y., and Zhang, H., (2024) "Domain-Driven Design for Microservices: An Evidence-Based Investigation," IEEE Transactions on Software Engineering, 50(6): 1425-1449. Available: https://doi.org/10.1109/TSE.2024.3385835.

[23] Trabelsi, I., Abdellatif, M., Abubaker, A., Moha, N., Mosser, S., Ebrahimi-Kahou, S., and Gueheneuc, Y. G., (2023) "From Legacy to Microservices: A Type-Based Approach for Microservices Identification Using Machine Learning and Semantic Analysis," Journal of Software: Evolution and Process, 35(10): e2503. Available: https://doi.org/10.1002/smr.2503.

[24] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H., (2024) "Large Language Models for Software Engineering: A Systematic Literature Review," ACM Transactions on Software Engineering and Methodology, 33(8): 1-79. Available: https://doi.org/10.1145/3695988.

[25] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., and Wang, Q., (2024) "Software Testing With Large Language Models: Survey, Landscape, and Vision," IEEE Transactions on Software Engineering, 50(4): 911-936. Available: https://doi.org/10.1109/TSE.2024.3368208.

[26] Nguyen, T.-T., Vu, T. T., Vo, H. D., and Nguyen, S., (2025) "An Empirical Study on Capability of Large Language Models in Understanding Code Semantics," Information and Software Technology. Available: https://doi.org/10.1016/j.infsof.2025.107780.

[27] Danyaro, K. U. et al., (2025) "LLM-Based Code Generation: A Systematic Literature Review With Technical and Demographic Insights," IEEE Access, 13: 194915-194939. Available: https://doi.org/10.1109/ACCESS.2025.3631952.

APPENDIX: CLAIM-SOURCE VALIDATION TABLE

Manuscript Claim	Reference	Original Metric	Interpretation Used
29–41% LLM semantic misclassification rate	Nguyen et al. [26]	Without context: 41% misclassification; with context: 29% misclassification (Table 3, equivalent-program discrimination results)	Range reported directly from study's context-varied conditions
34% inter-service coupling reduction	Zhong et al. [22]	34% reduction in inter-service coupling under full bounded-context contract coverage (Table 6, multi-system aggregate)	Reported as-is from aggregate result
48 systems studied	Zhong et al. [22]	48 microservice-based systems in empirical dataset	Reported as-is

0.85 contract-coverage threshold	Zhong et al. [22]	Coupling reduction observed at >= 85% contract coverage	Used as gate threshold; requires calibration
0.70 coherence threshold	Mazlami et al. [4]	Bounded-context decompositions with >= 70% intra-context coupling ratio produced fewer post-migration defects	Used as V1 threshold; requires calibration
k = 3 remediation cycles	Ziftci et al. [20]	Google study reports bounded retry before human escalation; exact cycle count not stated as 3	k = 3 is a design proposition inspired by, not directly derived from, the Google study
CODEMENV: 26.50% average pass@1; 43.84% GPT-4o	Cheng et al. [16]	Reported in abstract and Table 2 of CODEMENV paper	Reported as-is

Note: The k = 3 remediation cycle limit is a design proposition. The Google migration study [20] supports bounded retry logic with human escalation but does not specify a cycle count of three. This value should be recalibrated based on empirical remediation data from the prospective experiment in Section 8.2.