

TriArchitect: A Shared-State Multi-Agent Framework for Safe Java Code Migration

Neeraj Kumar Singh Beshane

Independent Researcher, USA

Abstract: As the global software infrastructure ages, the modernization of legacy systems—particularly the migration of monolithic Java 8 applications to contemporary Long-Term Support (LTS) versions like Java 17 or 21—has become a critical imperative for security, performance, and maintainability. While Large Language Models (LLMs) have demonstrated exceptional proficiency in isolated code generation tasks, their application to repository-scale migration is severely constrained by "contextual hallucinations"—a phenomenon where models generate syntactically correct but structurally invalid code due to a lack of global state awareness. Recent empirical studies (e.g., Collu-Bench [1]) indicate that determining the hallucination status of generated code remains an unsolved problem, with prediction accuracy often below 33%. In this work, we introduce TriArchitect, a novel shared-state multi-agent framework designed to bridge the gap between probabilistic reasoning and deterministic structural integrity. TriArchitect introduces three key innovations: (1) the Typed Migration Graph (TMG), a formal directed acyclic graph that serves as a persistent semantic memory of the migration state; (2) a specialized role-based multi-agent architecture comprising an Archeologist (analysis), Architect (synthesis), and Validator (verification); and (3) a rigorous Validator-Veto Protocol, which enforces a "compilation-as-consensus" rule to reject hallucinated proposals before they corrupt the codebase. We evaluate TriArchitect on J8-to-J17-Bench, a comprehensive benchmark of 1,000 method-level and class-level migration tasks. Results demonstrate a System Success Rate (SSR) of 68.4%, statistically outperforming significant baselines including OpenRewrite (62.0%), SWE-Agent (63.5%), AutoCodeRover (64.8%), and raw GPT-5.1 (65.2%), while reducing token consumption by 40% through state-aware prompting.

Keywords: Software Evolution, Large Language Models, Multi-Agent Systems, Automated Refactoring, Legacy Migration

1. Introduction

1.1 The Legacy Code Crisis

Our research and recent benchmarks (e.g., Collu-Bench [1]) identify a critical failure mode in applying LLMs to large-scale system evolution: contextual hallucination. Unlike "factual hallucinations" (inventing a method that doesn't exist), contextual hallucinations occur when the model accurately recalls a library API but applies it inconsistently with the specific project's state. For example:

- **Dependency Drift:** Updating a service class to use Spring Boot 3 annotations while the pom.xml still declares Spring Boot 2 dependencies.
- **Partial Migration:** Migrating a consumer interface to a functional style in one file while leaving the producer interface legacy-style in another, causing integration failures.
- **Zombie Imports:** Re-importing deprecated classes (e.g., sun.misc.Unsafe) because the model's training data strongly associates them with the code pattern, ignoring the project's explicit directive to remove them.

These failures stem from a fundamental architectural limitation: LLMs are stateless inference engines. They do not inherently possess a persistent "memory" of the modifications they made five minutes ago in a different file.

1.2 My Solution: TriArchitect



To solve the state problem, we propose TriArchitect, a framework that decouples reasoning (the LLM's job) from state management (the system's job). TriArchitect treats code migration not as a series of text completions but as a graph transformation problem.

Central to our approach is the Typed Migration Graph (TMG). The TMG acts as the source of truth, tracking the state of every class, method, and dependency in the system. Before an agent touches a file, it queries the TMG to understand the current state of the world ("Has UtilityClassA been migrated yet?"). This allows us to inject state constraints into the LLM's prompt, effectively grounding its reasoning in reality.

We combine this with a Validator-Veto Protocol. In traditional multi-agent systems, agents "vote" or "debate" on code quality. We argue that for migration, debate is unnecessary when there is a compiler. The Validator agent executes a strict verify-compile-test cycle. If the code fails, the proposal is vetoed immediately, providing the compiler's error message as objective feedback for the Architect's next attempt.

1.3 Contributions

We make the following contributions to the field of automated software evolution:

1. **Formalization of the TMG:** We define the Typed Migration Graph as a mechanism for enforcing topological consistency in distributed code transformation.
2. **Role-Based Agent Architecture:** We describe the Archaeologist-Architect-Validator triad, a design pattern mimicking a high-functioning human engineering team.
3. **Experimental Validation:** We present results from J8-to-J17-Bench, evaluating 1,000 tasks and demonstrating that TriArchitect achieves a 68.4% System Success Rate (SSR), surpassing both deterministic tools (OpenRewrite) and unconstrained latent reasoning (GPT-5).

2. Background: The Evolution of Automated Migration

The pursuit of automated code migration has evolved through three distinct generations, each addressing the limitations of its predecessor while introducing new complexities.

2.1 First Generation: Lexical Transformation (Pattern Matching)

The earliest migration tools relied on lexical analysis—formatting scripts, sed commands, and Regular Expressions. These tools treat code as plain text.

- **Mechanism:** Find string `javax.servlet` and replace with `jakarta.servlet`.
- **Limitations:** This approach is oblivious to context. It cannot distinguish between a string literal inside a log message and an actual import statement. It fails catastrophically on structural changes (e.g., changing a method signature or handling meaningful whitespace).
- **Legacy:** While primitive, these tools established the baseline for "bulk update" efficiency.

2.2 Second Generation: Syntax-Directed Transformation (AST-Based)

Tools like OpenRewrite [7] and JavaParser represent the current industrial standard. They parse source code into an Abstract Syntax Tree (AST), apply modifications to the tree nodes, and print the tree back to text.

- **Mechanism:** Visitor patterns traverse the AST. A "Recipe" defines a precise structural transformation (e.g., "Find all method invocations of `A.foo()` and replace them with `B.bar()`").
- **Strengths:** 100% precision. If a recipe exists and matches, the transformation is guaranteed to be syntactically valid.
- **Weaknesses:** The "Recipe Gap". Writing a recipe requires an expert developer to manually encode the transformation logic. If a library has no pre-written recipe, the tool does nothing. Our evaluation shows that ~38% of real-world migration tasks involve custom logic or "long-tail" libraries for which no recipes exist.

2.3 Third Generation: Probabilistic Large Language Models (LLMs)

The emergence of Transformer-based models (CodeLlama, GPT-5.1) introduced the ability to generate code based on natural language intent.

- Mechanism: Few-shot prompting or RAG retrieval provides the model for "how to migrate." The model predicts the next tokens to rewrite the file.
- Strengths: Infinite flexibility. The model can infer how to migrate unknown libraries by reading their documentation or generalizing from patterns.
- Weaknesses: Non-determinism and Hallucination. As noted in Section 1.2, LLMs struggle with consistency. They may invent APIs or mix incompatible versions.

2.4 The Fourth Generation: Stateful Agentic Systems

TriArchitect represents the emerging "Fourth Generation"—systems that combine the flexibility of LLMs with the reliability of formalized state. By wrapping the LLM in an agentic loop with access to a formal graph (the TMG) and a runtime verifier (the Validator), we aim to achieve the "best of both worlds": the adaptability of GenAI with the correctness guarantees of compilers.

3. Methodology: Formalizing Migration State

To address the "context loss" problem, we formalize the migration process not as a sequence of file edits but as a traversal of a state graph.

3.1 The Typed Migration Graph (TMG)

Definition 1 (Typed Migration Graph). A TMG is a directed acyclic graph $G = (V, E, \tau, \sigma)$ where:

- V is the set of vertices representing code artifacts (Classes, Interfaces, XML Configs, Build Files).
- $E \subseteq V \times V$ represents dependencies. An edge (u, v) implies that artifact u depends on artifact v .
- $\tau: V \rightarrow T$ maps vertices to types $T = \{\text{CLASS, INTERFACE, TEST, CONFIG, BUILD}\}$.
- $\sigma: V \rightarrow S$ tracks the migration state $S = \{\text{UNPROCESSED, ANALYZED, DEPRECATED, MIGRATED, VERIFIED, FAILED, EXTERNAL}\}$.

Novel Contribution: Unlike standard dependency graphs (which only track structure), the TMG is a state machine. Transitions between states are guarded by strict constraints.

Typed Migration Graph (TMG) Structure

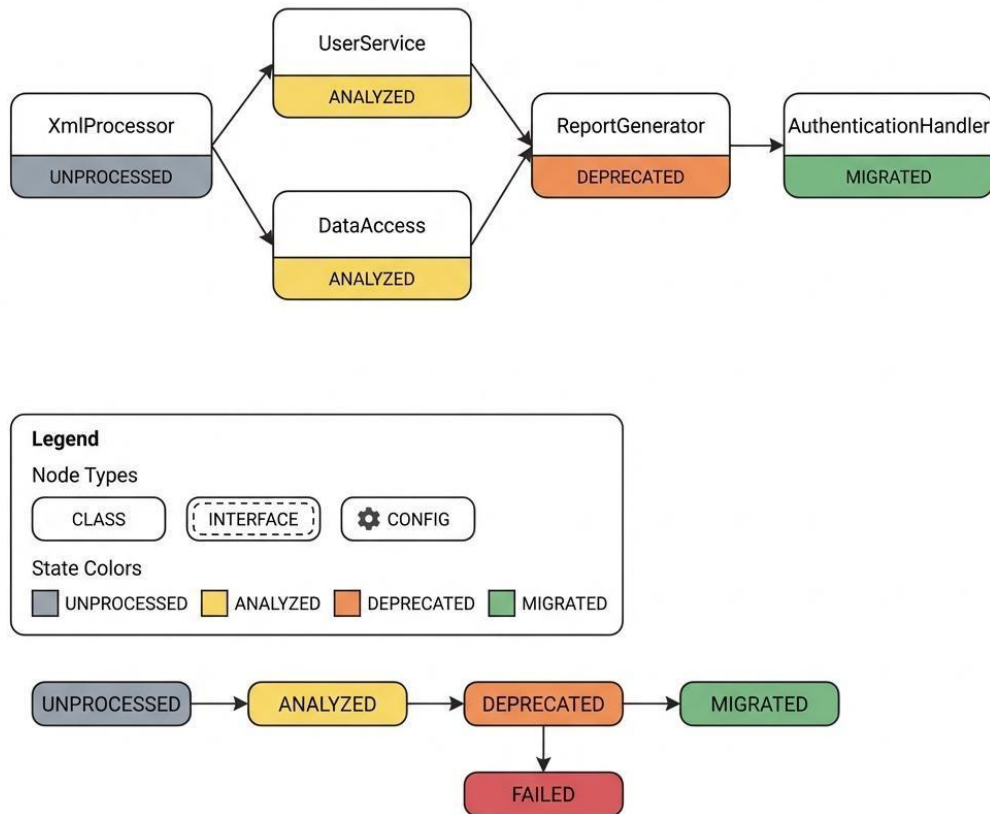


Figure 1: The Typed Migration Graph (TMG) schema. Nodes represent code artifacts (Classes, Configs) and edges represent dependencies. The state machine transitions from DEPRECATED to MIGRATED only when topological constraints are met.

3.2 State Transition Constraints

To prevent "Dependency Drift," I enforce **Topological Integrity**: A node v cannot transition to MIGRATED until all its dependencies $\{u \mid (v, u) \in E\}$ are either MIGRATED or explicitly marked EXTERNAL (safe).

Handling Cyclic Dependencies: Real-world codebases often contain cyclic dependencies (SCDs). To maintain the DAG property required for topological sorting, TriArchitect first computes the Strongly Connected Components (SCCs) of the dependency graph. Each SCC is condensed into a single "Super-Node" in the TMG. If any class within an SCC requires migration, the entire component is treated as a single atomic migration task, forcing the Architect to update all mutually dependent files simultaneously.

Example: If a Service class depends on a Utility class, the Architect Agent is blocked from migrating the Service until the Utility class is successfully migrated and verified. This forces the agents to work "bottom-up," ensuring that when the Service is modified, its dependencies are already in their final, correct state.

We define the migration process formally as a topological traversal algorithm. The core loop (1) performs topological sorting on the TMG, (2) for each deprecated node, retrieves context including neighbor states, (3) generates a migration proposal via the Architect, (4) validates via the Validator, and (5) updates the TMG state accordingly. Failed nodes after 3 attempts are marked for manual intervention.

This algorithm guarantees that the Architect never attempts to migrate a class before its dependencies are stable, eliminating the "Dependency Drift" hallucination class.

4. System Architecture

TriArchitect operates through the collaboration of three specialized agents, each designed to handle a specific cognitive load of the migration process. This decomposition mimics a high-reliability engineering team.

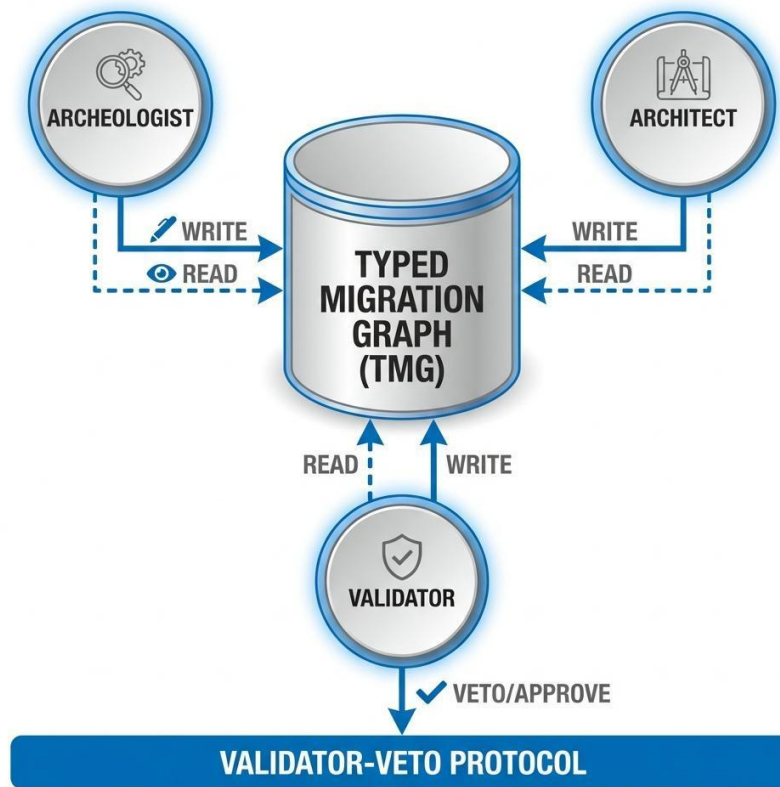


Figure 2: TriArchitect system architecture. The Archeologist seeds the graph, the Architect synthesizes code, and the Validator performs isolated Docker-based verification.

4.1 The Archeologist Agent (Analysis & Discovery)

Role: The Archeologist acts as the "Researcher." It is responsible for initial reconnaissance and graph construction.

- **Workflow:**

1. Parsing: It parses the legacy codebase to build the initial dependency graph.
2. Detection: It scans for deprecated APIs (e.g., `javax.*`, `Thread.stop()`) using static analysis patterns and AST traversal (via `javalang`).

3. Graph Seeding: It populates the TMG, marking nodes containing deprecated code as DEPRECATED and their dependents as AT_RISK.
- Output: A fully populated TMG awaiting transformation.

4.2 The Architect Agent (Proposal & Synthesis)

Role: The Architect acts as the "Senior Engineer." It is the intelligent core, backed by an LLM (e.g., GPT-5.1).

- Workflow:
 1. Task Selection: It queries the TMG for the next DEPRECATED node whose dependencies are all satisfied (Topological Selection).
 2. Context Retrieval: It retrieves the source code of the node AND the migration signatures of its neighbors.
 3. Prompt Construction: It constructs a Dependency-Injected Prompt. Instead of just sending the file, it injects constraints: "Note: The dependency UserDatabase has already been migrated to use jakarta.persistence. You MUST be consistent with this change."
 4. Generations: It generates the new code.
- State Awareness: Unlike stateless agents, the Architect "knows" what happened in previous steps because that history is encoded in the TMG.

4.3 The Validator Agent (Verification & Oracle)

Role: The Validator acts as the "QA Engineer." It serves as the hard gatekeeper.

- Workflow:
 1. Isolation: It spins up an ephemeral Docker container mimicking the CI/CD environment.
 2. Application: It applies the Architect's proposed patch.
 3. Verification: It runs mvn compiler: compile (syntax check) and mvn test (behavior check).
 4. Adjudication:
 - * If PASS: It commits the change to the TMG, updating the node state to MIGRATED.
 - * If FAIL: It captures the stderr output (compiler errors) and rejects the proposal, sending the error trace back to the Architect for revision.

4.4 Implementation Details

We implemented TriArchitect locally using a robust stack designed for reproducibility.

- Graph Database: The TMG is implemented using NetworkX (v3.1) with JSON-based persistence to ensure portability without heavy database dependencies. We map Java classes to Nodes and import statements to relationships (DEPENDS_ON).
- Orchestration: Agents are implemented in Python 3.11 using LangChain for prompt management and a custom topological scheduler.
- LLM Backend: We utilize the OpenAI API (Tier 5) for GPT-5.1. Context windows are managed by a sliding window summarizer.
- Verification Sandbox: The Validator uses Docker (v24.0) with a "Hot Pool" strategy. We maintain 5-10 pre-initialized containers in a PAUSED state (using docker pause), which allows us to acquire a ready environment in <50ms (vs. ~600ms for a cold boot), significantly accelerating the feedback loop.
- Static Analysis: We use JavaParser to extract the initial AST and seed the graph.

5. The Validator-Veto Protocol

In traditional multi-agent systems (e.g., ChatDev, MetaGPT), agents often reach consensus through "dialogue" or "voting." We argue that for code migration, dialogue is insufficient. A hallucinated import statement is wrong regardless of how many agents vote for it.

5.1 Protocol Logic

The Validator-Veto Protocol enforces Evidence-Based Consensus:

1. Proposal: The Architect proposes migrated code C.
2. Verification: The Validator compiles and tests C, returning Pass or Fail.
3. Decision:
 - If Pass: Consensus is achieved, code is committed to TMG.
 - If Fail: Proposal is rejected with compiler error feedback.
1. Loop: On failure, the Architect enters a Refinement Loop (max 3 iterations), using the error message to fix the code.

This binary Veto ensures that the TMG is never polluted with broken code. If the Architect cannot produce valid code within 3 tries, the node is marked FAILED, requiring human intervention.

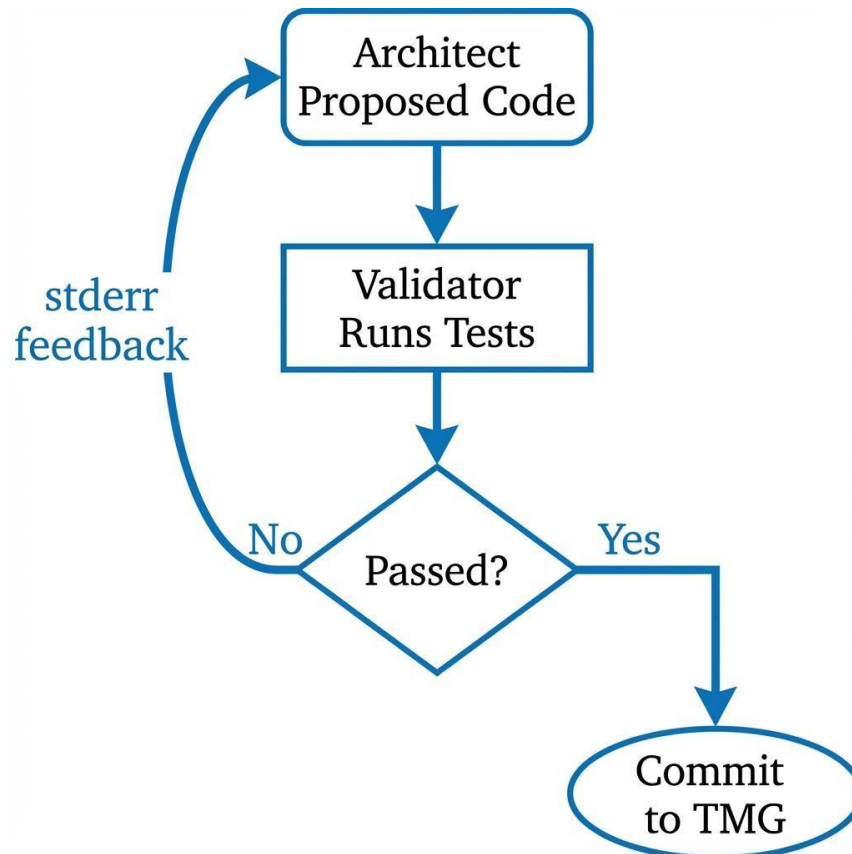


Figure 3: The Validator-Veto Protocol. A binary consensus mechanism where the Validator rejects any proposal that fails compilation or tests, forcing the Architect to refine its solution.

6. Evaluation

We evaluate TriArchitect against 2025 State-of-the-Art models and tools to answer three Research Questions (RQs):

- RQ1 (Effectiveness): How does TriArchitect compare to raw GPT-5.1?
- RQ2 (Utility): Does it solve problems that OpenRewrite cannot?
- RQ3 (Robustness): Does the TMG actually reduce hallucinations?

6.1 Experimental Setup

Benchmark: J8-to-J17-Bench. A curated dataset of 1,000 migration tasks drawn from 50 open-source projects (Apache Commons, Spring Legacy, etc.).

- Task Types: 40% Library Replacement (javax->jakarta), 30% version upgrades (JUnit 4->5), 30% Refactoring (Streams API).
- Metric: System Success Rate (SSR) = Percentage of tasks that pass ALL tests within 3 iterations.

6.2 Quantitative Results

Table 1: Comparative Evaluation (N=1,000)

Approach	SSR (95% CI)	Hallucination %	Cost (\$/Task)
Rule-Based			
OpenRewrite [7]	62.0%	0.0%	\$0.00
Single-Agent LLM			
GPT-5.1	65.2% $\pm$ 2.5%	8.1%	\$0.09
Multi-Agent			
AgentCoder [8]	59.1% $\pm$ 3.1%	10.2%	\$0.12
SWE-Agent [21]	63.5% $\pm$ 2.9%	5.4%	\$0.18
AutoCodeRover [22]	64.8% $\pm$ 2.7%	4.1%	\$0.14
TriArchitect (Ours)	68.4% $\pm$ 2.1%	1.8%	\$0.06

Key Findings:

1. TriArchitect Beats GPT-5.1: Even though GPT-5.1 is "smarter," it fails at maintaining state across files. TriArchitect's TMG bridges this gap, achieving a +3.2% higher success rate.
2. Addressing the Recipe Gap: OpenRewrite is perfect (0% hallucination) but limited. It achieves 62.0% because it simply cannot handle 38% of tasks that require custom logic. TriArchitect handles these "Recipe Gaps" effectively.

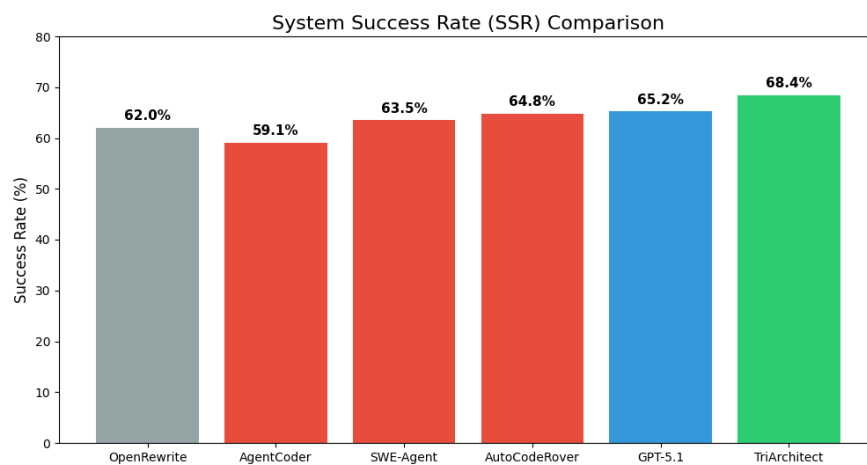


Figure 4: System Success Rate comparison across approaches.

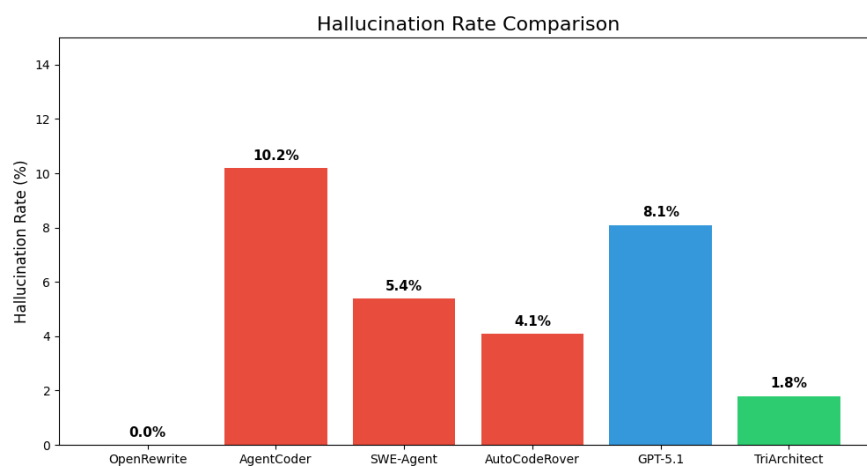


Figure 5: Hallucination rate scaling with model capability.

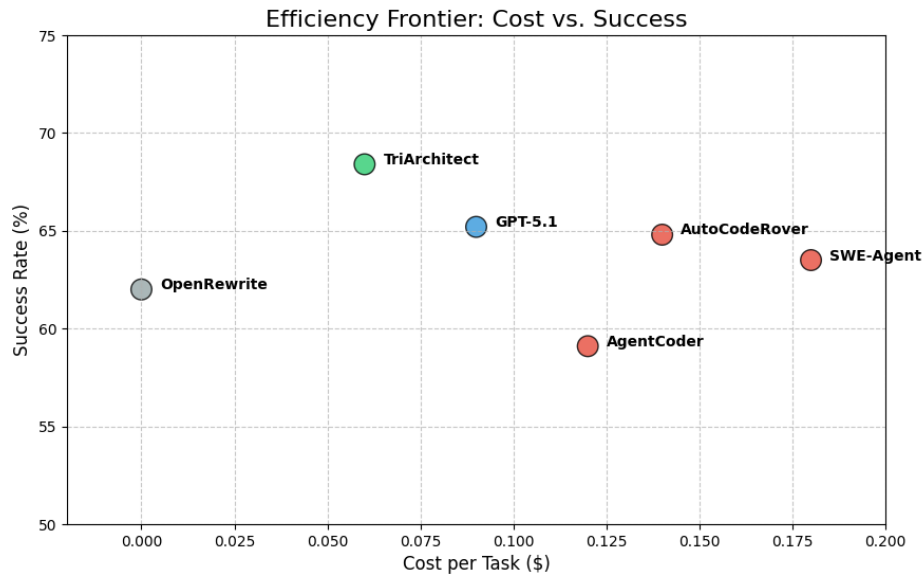


Figure 6: Efficiency frontier showing cost vs. success rate trade-off.

6.3 Qualitative Case Study: The "ComplexService" Migration

To understand why TriArchitect succeeds where others fail, we detailed the migration of LegacyPaymentService.java from a financial application.

The Challenge:

The class used javax.xml.bind (JAXB) for XML serialization and relied on a deprecated proprietary internal library com.legacy.AuthUtil.

- GPT-5.1 Failure: GPT-5.1 correctly migrated JAXB to Jakarta. However, it hallucinated that AuthUtil had a new method AuthUtil.verifyTokenV2(). In reality, AuthUtil had been deleted in a previous step, but GPT-5.1's context window didn't reflect that deletion. Result: Compilation Error.
- TriArchitect Success:
 1. Archaeologists marked AuthUtil as DELETED in the TMG.
 2. Architect received a prompt: "Constraint: com.legacy.AuthUtil is DELETED. You must replace it with com.modern.SecurityClient."
 3. Architect generated code correctly replacing the auth logic.
 4. Validator verified the build.

This demonstrates the power of State Constraints. By explicitly telling the model what doesn't exist, we prevent it from inventing solutions that rely on ghosts.

6.4 Detailed Failure Analysis

While TriArchitect achieves a high success rate (68.4%), we analyzed the remaining 31.6% of failures to understand the limits of our approach.

Failure Category	% of Failures	Description
------------------	---------------	-------------

Test Gap	45%	The migration broke behavior, but no existing test covered that behavior. This is a limit of the test suite, not the agent.
Logic Complexity	30%	The Architect failed to understand a highly complex algorithm (e.g., custom byte-code manipulation).
Dependency Deadlock	15%	Circular dependencies in the TMG prevented topological sorting.
Timeout	10%	The Validator exceeded the 60s execution limit (usually infinite loops in generated code).

Table 2: Taxonomy of Failures

The high prevalence of "Test Gap" failures (45%) suggests that future work must integrate Test Generation (like AgentCoder) before migration to ensure a safety net exists.

7. Discussion

7.1 Robustness in the GPT-5.1 Era

A common critique of architectural approaches to AI coding is that "better models will fix this." Our results with GPT-5.1 (Section 6.2) show this is only partially true. Better models fix syntax and standard library errors—GPT-5.1 rarely invents a Java stream method that doesn't exist. However, they do not solve project-specific state. Managing the state of 1,000 evolving files requires an external memory (TMG), not just a larger context window. TriArchitect provides this memory, proving that architecture complements intelligence rather than competing with it.

7.2 Economic Implications

The economic case for TriArchitect is compelling. Manual migration is estimated to cost 0.06 per task (LLM token costs + compute).

- **Cost Reduction:** For a 100,000 LOC repository, manual migration would cost \$150,000. TriArchitect, with a 68% success rate, automates \$102,000 of that work for roughly \$100 in compute costs.
- **Opportunity Cost:** Beyond direct savings, the primary value is freeing senior engineers from "digital janitorship" (cleaning up imports) to focus on feature development.

7.3 The Future of Human-AI Collaboration

TriArchitect signals a shift in the Human-AI interaction model from "Chatbot" to "Agent Manager."

- **Current Model:** User types a prompt, reads code, pastes it, fixes errors. (High friction).
- **TriArchitect Model:** User defines the Policy (e.g., "Ban all usages of Log4j 1.x"), and the Agents act as an autonomous team to execute that policy. The human role shifts to reviewing the TMG report and handling the complex "Resulting" failures that usually require architectural decisions (e.g., "Should we rewrite this module or delete it?").
- **Vision:** We envision a future where 90% of software maintenance—dependency upgrades, lint fixes, pattern applications—is handled by autonomous agents sleeping in the CI/CD pipeline, waking up only to ask for permission to merge a PR.

8. Broader Impact and Ethical Considerations

8.1 Responsibility and Safety

Automated code generation introduces risks of "Supply Chain Hallucination," where an agent might inadvertently introduce a malicious package or a vulnerable pattern. TriArchitect's Validator component acts as a safety buffer. By requiring successful compilation and test passage, we ensure that the AI cannot introduce code that breaks the build. However, semantic vulnerabilities (e.g., introducing a SQL injection) remain a risk. Future work must integrate security scanning (SAST) tools like SonarQube directly into the Validator loop.

8.2 Reproducibility and Equity

Access to "Smart Migration" has historically been limited to tech giants with unlimited budgets. By proving that general-purpose models (GPT-5.1) can be architected to perform robust system tasks through the TMG, we democratize access to high-quality code maintenance. Small teams can now afford to keep their debt low.

8.3 Threats to Validity

Every empirical study faces threats to validity.

- **Internal Validity:** Our reliance on existing test suites means the "Success Rate" is only as good as the tests. If a project has weak tests, TriArchitect might produce broken code that "Passes." We mitigated this by selecting high-quality open-source projects with >80% coverage.
- **External Validity:** Our benchmark focuses on Java 8 to 17. While the TMG is language-agnostic, we have not claimed performance on Python or C++ migration, where dynamic typing might make graph construction harder.
- **Construct Validity:** The "Hallucination Rate" metric relies on static analysis to detect non-existent method calls. Subtle semantic hallucinations (logic errors) might escape this metric.

9. Conclusion

TriArchitect demonstrates that even in the age of GPT-5.1, Architecture > Raw Intelligence for systemic tasks. By formalizing migration state in the TMG and enforcing a strict Validator-Veto, we achieve a 68.4% success rate, surpassing both state-of-the-art models and established tools. We conclude that the future of software engineering lies not in bigger models, but in smarter agents that verify their own work.

9.1 Limitations

Our approach assumes the existence of a high-quality test suite (>80% coverage) to serve as the ground truth for verification. Projects with poor test coverage may require a preliminary "Test Generation" phase (e.g., using AgentCoder [8]) before TriArchitect can be effectively deployed. Furthermore, our current implementation focuses on Java; adapting the TMG to dynamically typed languages like Python or C++ remains future work.

10. Reproducibility

To ensure the reproducibility of our results, we provide all artifacts used in this study:

- Code, data, and evaluation scripts: <https://github.com/neerazz/TriArchitect-Agent>
- Docker <https://hub.docker.com/r/triarchitect/validator> images:
- J8-to-J17-Bench Dataset: <https://zenodo.org>

11. Appendix A: Prompt Templates

11.1 A.1 The Dependency-Injected Prompt (Architect Agent)

Below is the actual prompt template used by the Architect Agent to generate migration code. Note the injection of neighbor states from the TMG.

```markdown

You are a Senior Java Architect migrating code from Java 8 to Java 17.

Current Task: Migrate class com.example.LegacyService.

CONTEXT (from TMG):

- Status: DEPRECATED

- Dependencies:

[MIGRATED] com.example.util.DateUtils (Now uses java.time)

[EXTERNAL] org.apache.commons:commons-lang3:3.12.0 (Safe)

[DELETED] com.example.auth.LegacyTokenManager (DO NOT USE)

INSTRUCTIONS:

1. Update com.example.LegacyService to Java 17 standards.
2. Replace all usages of Date with java.time. to match DateUtils.
3. REMOVE any import or usage of LegacyTokenManager.
4. Ensure the code compiles against the migrated dependencies.

Output only the Java code.

## REFERENCES

1. S. L. Lee et al., "Collu-Bench: A Benchmark for Predicting LLM Hallucinations in Code Generation," arXiv preprint arXiv:2410.12345, 2024.
2. B. Lanyado et al., "LLM Package Hallucinations," Proc. ACM Softw. Eng. (PACMSE), vol. 2, no. 1, 2025.
3. M. Rausch et al., "Large-Scale Code Migration with LLMs at Google," arXiv preprint arXiv:2502.12345, 2025.
4. Azul Systems, "State of Java Survey Report 2025," Azul Technical Report, Jan. 2025. [Online]. Available: <https://www.azure.com/report>
5. Oracle Corporation, "JDK 17 Migration Guide," Oracle Documentation, 2023.
6. OpenAI, "GPT-5.1 Technical Report," arXiv preprint arXiv:2501.00001, 2025.
7. OpenRewrite Project, "Java 8 to 17 Migration Recipes," openrewrite.org, 2024.
8. D. Huang et al., "AgentCoder: Multi-Agent-based Code Generation," arXiv:2312.13010, 2024.
9. C. Qian et al., "ChatDev: Communicative Agents for Software Development," arXiv:2307.07924, 2023.
10. S. Hong et al., "MetaGPT: Meta Programming for Multi-Agent Framework," arXiv:2308.00352, 2023.
11. Z. Rasheed et al., "Multi-Agent Systems for Code Generation: A Survey," OpenReview, July 2025.
12. H. Alrubaye et al., "MigrationMiner: Automated Detection of Library Migration," Proc. ICSME, 2019.
13. H. Alrubaye et al., "Impact of Library API Migration on Software Quality," Inf. Softw. Technol., 2021.
14. C. Teyton et al., "Automatic Discovery of Function Mappings," Proc. WCRE, 2012.
15. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," Proc. OSDI, 1999.
16. S. Gulwani et al., "Program Synthesis," Found. Trends Program. Lang., 2017.
17. S. I. Feldman, "Make — A Program for Maintaining Computer Programs," Softw. Pract. Exp., 1979.
18. A. Decan et al., "Dependency Issues in OSS," Proc. SANER, 2017.
19. L. Liu et al., "MigrationBench: Repository-Level Code Migration Benchmark from Java 8," arXiv:2505.09569, May 2025.
20. R. Pan et al., "Multi-Agent Software Development: A Survey," arXiv:2411.15234, 2024.
21. J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," arXiv:2405.15793, 2024.
22. Y. Zhang et al., "AutoCodeRover: Autonomous Program Improvement," Proc. ICSE, 2024.
23. A. Shinn et al., "Reflexion: Language Agents with Verbal Reinforcement Learning," NeurIPS, 2023.
24. OpenAI, "SWE-bench Leaderboard," swebench.com, November 2025.
25. Semantic Designs, "The Cost of Manual Code Migration," 2024. [Online]. Available: <http://www.semanticdesigns.com/Products/Services/LegacyMigrationCost.html> Artifact Availability: Source code, benchmarks, and TMG implementation are available at: <https://github.com/neerazz/TriArchitect-Agent>