Cross-Language Code Migration Using Multi-Agent Systems: Evaluating Small-Scale Agentic Program Translation

James Freeman
Pembroke College, University of Oxford
james.freeman@pmb.ox.ac.uk

December 2025

Abstract

This paper evaluates LLM-based multi-agent systems for autonomous cross-language code migration using a 352-line Python codebase migrated to Rust, Java, and Go. Unlike AI-assisted approaches requiring human guidance, our migrations run end-to-end without human intervention: agents read source code, generate target implementations, execute quality gates, and iterate on failures autonomously. We performed six migrations—two strategies (module-by-module and feature-by-feature) across three target languages—with automated metrics collection including cost, duration, lines of code, and test coverage. All migrations completed successfully and passed quality gates. Observed costs ranged from \$6.43 to \$14.11 USD per migration; test coverage ranged from 64.9% to 94.8% depending on target language and strategy. A notable finding: while the source Python had a 0.6x test-to-production ratio, agents generated 2–4x more test code than production code in target languages (e.g., Go MbM: 3,207 test LOC for 856 production LOC)—a 3–6x increase in relative test investment. Rust migrations achieved the highest coverage (94–95%), followed by Java (73–84%) and Go (65–68%). Module-by-module completed faster (32–37 minutes) than feature-by-feature (55–60 minutes), but cost differences showed no consistent pattern. The four-phase methodology—I/O contract generation, source analysis, sequential migration, and behavioral review—proved language-agnostic. These results establish that automated migration is achievable for small-scale, well-structured codebases, while highlighting the need for multiple runs to characterize performance distributions.

1 Introduction

Cross-language code migration represents a significant challenge in software engineering. Organizations frequently need to port codebases between languages for performance optimization, ecosystem alignment, or maintainability improvements. Traditional approaches require substantial manual effort, with developers reading source code, understanding its semantics, and rewriting equivalent implementations in the target language.

Recent advances in Large Language Models have enabled automated code generation at unprecedented quality levels. However, applying these capabilities to complete codebase migration introduces challenges beyond single-function generation. A successful migration must preserve not only the public API but also the precise behavioral semantics of the original implementation, including edge cases and error handling.

This paper presents evidence that LLM-based multi-agent systems can reliably automate cross-language migration for a specific class of codebases: small-scale systems with low complexity, well-defined module boundaries, and strong test coverage. Through experimental runs migrating Python to Rust, Java, and Go, we developed and validated a four-phase methodology that achieves 100% behavioral equivalence across multiple target languages. The successful migration to three structurally different languages—Rust's ownership model, Java's garbage collection, and Go's composition-based design—demonstrates that our methodology is language-agnostic rather than target-specific.

1.1 Problem Statement

Existing LLM-based migration approaches typically operate in an AI-assisted mode: developers prompt models for code suggestions, manually review and integrate outputs, and handle compilation errors themselves. This human-in-the-loop approach limits throughput and requires developer attention throughout the process.

Traditional LLM-based code generation tools suffer from fundamental limitations when applied to fully autonomous migration. First, most tools lack file system access, requiring developers to manually copy source code into prompts. This approach does not scale to multi-file codebases and prevents agents from exploring dependencies. Second, without build tool integration, generated code cannot be verified for correctness until manual compilation. Third, single-shot generation provides no mechanism for iterative refinement based on compilation errors or test failures.

Perhaps most critically, existing approaches conflate API compatibility with behavioral equivalence. A migration that preserves function signatures may still produce different outputs for identical inputs, particularly in edge cases involving operator precedence, associativity, or error handling.

1.2 Research Questions

This study addresses three primary research questions:

1. **Feasibility**: Can multi-agent LLM systems automate complete cross-language migration with verified behavioral equivalence?

2. **Generality**: Does a migration methodology developed for one target language transfer to structurally different target languages?

3. **Scope**: What characteristics of source codebases determine whether automated migration is viable?

1.3 Contributions

This paper makes four contributions:

1. **Autonomous migration**: We demonstrate end-to-end migration without human intervention, where agents autonomously handle error recovery through feedback loops with compilers, linters, and test runners.

2. **Multi-language validation**: We show the same four-phase methodology succeeds for Rust, Java, and Go, establishing language-agnostic generality.

3. **Methodology specification**: We define a repeatable four-phase process (I/O contract, analysis, migration, review) with explicit quality gates.

4. **Scope characterization**: We identify the characteristics that made our subject system amenable to autonomous migration, informing future scaling research.

2 Background

2.1 Subject System

Our experiments used `rpn2tex`, a command-line tool that converts Reverse Polish Notation mathematical expressions to LaTeX format. The Python implementation comprises 352 lines of production code across seven modules: token definitions, abstract syntax tree nodes, error handling, lexical analysis, parsing, LaTeX generation, and command-line interface.

We deliberately selected a *trivial* subject system to establish baseline feasibility before investigating more complex scenarios. The system exhibits characteristics that represent ideal conditions for automated migration:

- **Low cyclomatic complexity**: Each function implements straightforward control flow without deep nesting or complex branching.

- **Clear module boundaries**: Dependencies flow unidirectionally from tokens through lexer, parser, and generator.

- **No external dependencies**: The codebase uses only Python standard library features.
- **Deterministic behavior**: Identical inputs always produce identical outputs.
- **Explicit error handling**: Error conditions are well-defined with position tracking.
- **Comprehensive testability**: Input-output behavior is easily captured and verified.

2.1.1 Code Metrics

The source Python implementation contains 352 lines of production code and 204 lines of test code across 7 modules. Table 1 presents lines of code for all migrations, distinguishing production code from test code.

Table 1: Lines of code by target and strategy

Target	Strategy	Prod	Test	Ratio
Python	(source)	352	204	0.6x
Rust	MbM	601	1,070	1.8x
Rust	FbF	513	1,354	2.6x
Java	MbM	553	1,805	3.3x
Java	FbF	516	1,022	2.0x
Go	MbM	856	3,207	3.7x
Go	FbF	479	1,380	2.9x

A notable finding: while the source Python had a 0.6x test-to-production ratio (204 test LOC for 352 prod LOC), agents generated 2–4x more test code than production code in the target languages. The test-to-production ratio ranged from 1.8x (Rust MbM) to 3.7x (Go MbM)—a 3–6x increase in relative test coverage compared to the source. This suggests the quality gates and I/O contract requirements drive substantial test generation as agents work to satisfy behavioral validation criteria.

These characteristics represent a best-case scenario for automated migration. The research question is not whether this particular system can be migrated—it clearly can—but whether the methodology that succeeds here generalizes to other target languages and, eventually, to more complex systems.

2.2 Multi-Agent Architecture

All experiments employed the Claude Agent SDK, which enables spawning specialized subagents with different tool access and model configurations. The architecture consists of an orchestrating agent that coordinates specialized workers.

The analyst agent uses a lightweight model with read-only file access for codebase analysis. The migrator agent uses a more capable model with full tool access including file writing, editing, and shell command execution. The reviewer agent uses a lightweight model to validate generated code against specifications.

This separation allows cost optimization by using expensive models only for code generation while using cheaper models for analysis and validation tasks.

2.3 Quality Gates

Each module migration required passing four quality gates before proceeding:

1. **Compilation**: The target language compiler must accept the code without errors (rustc, javac, or go build).
2. **Linting**: The language-specific linter must report zero warnings in strict mode (Clippy for Rust, Checkstyle for Java, go vet for Go).
3. **Formatting**: The code formatter must not require any changes (rustfmt, google-java-format, gofmt).
4. **Testing**: All unit tests must pass.

These gates ensure that generated code meets production quality standards and that issues are caught and corrected during migration rather than discovered later. The autonomous feedback loop between code generation and quality gates allows agents to iterate on failures without human intervention.

3 Method

3.1 Four-Phase Migration Process

Our methodology employs a four-phase process designed to ensure behavioral equivalence:

Phase 0: I/O Contract Generation. Before migration begins, the source implementation is executed on a curated set of test inputs covering basic operations, operator precedence, associativity, edge cases, and error conditions. The exact outputs are captured as a behavioral contract that target implementations must satisfy.

Phase 1: Source Analysis. An analyst agent reads all source files and produces a comprehensive migration specification document. This document captures module structure, dependencies, public APIs, and implementation details in a format optimized for migration agents.

Phase 2: Sequential Migration. Migrator agents receive the specification document and I/O contract, generating target language implementations for each module. Each module must pass quality gates (compilation, linting, formatting, tests) before proceeding.

Phase 3: Behavioral Review. Reviewer agents validate that generated code satisfies the I/O contract by executing all test cases and comparing outputs to the captured contract.

3.2 Key Design Decisions

Two design decisions proved critical during methodology development:

Focused agent contexts. Early experiments with embedding source code directly in prompts performed poorly—four times slower and 38% more expensive than focused approaches. Large contexts caused response latency (single responses taking 20+ minutes) and agents ignored embedded content, performing redundant file operations. The multi-phase approach keeps each agent's context minimal.

Behavioral contracts over API compatibility. Initial migrations that passed all quality gates still exhibited 19% behavioral discrepancies in output formatting. For example, the input `5 3 - 2 -` produced `$5 - 3 - 2$` in Python but `$( 5 - 3 ) - 2$` in early Rust migrations—mathematically equivalent but semantically different. The I/O contract phase eliminated these discrepancies.

3.3 Metrics

We measured wall-clock duration, API cost in US dollars, production lines of code, test coverage (line and function/method), and I/O contract match rate (percentage of test cases producing identical output to the source implementation).

4 Results

4.1 Migration Outcomes

We performed migrations to three structurally different target languages: Rust (a systems language with ownership semantics), Java (a managed language with garbage collection), and Go (a systems language with composition-based design and explicit error handling). We tested both module-by-module and feature-by-feature strategies for each target. All migrations achieved 100% behavioral equivalence on the 21-case I/O contract. Table 2 summarizes the results.

Table 2: Migration results by target and strategy

Target	Strategy	Dur.	Cost	Msgs	Cov.
Rust	MbM	32m	\$8.83	906	94.7%
Rust	FbF	60m	\$9.60	779	94.8%
Java	MbM	37m	\$14.11	1370	84.0%
Java	FbF	55m	\$8.18	779	73.0%
Go	MbM	37m	\$8.99	1009	64.9%
Go	FbF	56m	\$6.43	853	68.2%

All six migrations completed successfully and passed quality gates. Module-by-module was consistently faster (32–37 minutes) than feature-by-feature (55–60 minutes). Cost varied without clear pattern: Java module-by-module was the most expensive run (\$14.11), while Go feature-by-feature was the least expensive (\$6.43). Test coverage varied substantially by target language: Rust achieved 94–95%, Java 73–84%, and Go 65–68%.

Limitations of single-run observations: Each configuration was executed once. The migration process involves non-deterministic model behavior, and observed differences may reflect run-to-run variance rather than systematic strategy effects. The Java module-by-module outlier (\$14.11, 1370 messages) may indicate error recovery cycles or could simply be an unlucky run.

5 Discussion

5.1 Why Smaller Contexts Outperform

The counterintuitive finding that embedding source code degraded performance can be explained by several factors. In transformer-based models, attention complexity scales quadratically with context length. Larger contexts increase both latency per token and total tokens processed. When subagents inherit parent context, this cost multiplies across invocations.

Furthermore, large contexts dilute the signal of specific instructions. When source code is embedded alongside instructions, the model must attend to substantially more content to locate relevant information. A focused specification document provides higher signal-to-noise ratio.

Finally, agents exhibit emergent behavior that may not follow explicit instructions. Despite being told to use embedded content, agents still performed file operations. This suggests that behavioral constraints should be enforced through tool access rather than prompt instructions.

5.2 Importance of Behavioral Contracts

Migrations that pass all quality gates may still exhibit behavioral discrepancies. In our experiments, early migrations without I/O contracts showed 19% of test cases producing different output despite successful compilation, linting, and test passage. Auto-generated tests validate implementation self-consistency but cannot detect semantic drift from the source implementation.

Input-output contracts provide an oracle derived from the source implementation's actual behavior. By capturing exact outputs for representative inputs, contracts enable detection of behavioral differences that might otherwise go unnoticed. This is particularly important for edge cases involving operator precedence, associativity, and error handling where reasonable implementations may differ.

5.3 Migration Strategy Comparison

We evaluated two migration strategies on the same subject system: *module-by-module* (vertical slices) and *feature-by-feature* (horizontal slices).

Module-by-module migrates each source module completely before proceeding to the next. The order follows the dependency graph: tokens, AST, error handling, lexer, parser, generator, CLI. Each module is independently testable after migration.

Feature-by-feature migrates horizontal slices across all modules. Each feature (e.g., "addition operator") is migrated through lexer, parser, and generator before proceeding to the next feature. This strategy enables incremental I/O validation per feature.

Table 3 presents observed metrics for both strategies across target languages.

Table 3: Strategy comparison by target language

Target	Strategy	Cost	Dur.	Msgs
Rust	MbM	$8.83	32m	906
Rust	FbF	$9.60	60m	779
Java	MbM	$14.11	37m	1370
Java	FbF	$8.18	55m	779
Go	MbM	$8.99	37m	1009
Go	FbF	$6.43	56m	853

Both strategies produced correct output for all test cases. Module-by-module was consistently faster (32–37 minutes vs 55–60 minutes), but feature-by-feature produced fewer messages in most cases. Cost showed no consistent pattern by strategy.

The Java module-by-module run was the most expensive ($14.11) with the most messages (1370), potentially indicating more error recovery cycles.

We cannot draw conclusions about strategy superiority from single runs. Both strategies reliably produce correct output. For this trivial codebase, strategy selection does not affect correctness.

5.4 Tool Usage Patterns

Table 4 shows tool invocation counts across all migrations. These counts reflect how agents interact with the codebase during migration.

Java module-by-module used the most Bash invocations (334) and Read operations (184), consistent with its higher message count and cost. Go module-by-module notably used only 1 Edit operation versus 48 Write operations, suggesting a strategy of writing complete files rather than iterative refinement. Feature-by-feature migrations generally

Table 4: Tool invocations by target and strategy

Target	Strategy	Bash	Read	Write	Edit
Rust	MbM	237	123	30	14
Rust	FbF	178	108	24	47
Java	MbM	334	184	57	42
Java	FbF	157	119	37	31
Go	MbM	291	119	48	1
Go	FbF	196	131	35	17

showed higher Edit-to-Write ratios (e.g., Rust FbF: 47 Edit vs 24 Write), indicating more iterative file modification.

5.5 Test Coverage

Table 5 presents test coverage measurements for migrations where coverage tooling was available.

Table 5: Test coverage by target and strategy

Target	Strategy	Line Coverage
Rust	MbM	94.7%
Rust	FbF	94.8%
Java	MbM	84.0%
Java	FbF	73.0%
Go	MbM	64.9%
Go	FbF	68.2%

Rust migrations achieved the highest coverage (94–95%) regardless of strategy, measured using cargo-llvm-cov. Java coverage ranged from 73% to 84%, measured via JaCoCo. Go achieved the lowest coverage (65–68%), measured using Go's built-in coverage tooling.

The coverage disparity across languages warrants investigation. Rust's high coverage may reflect the language's emphasis on exhaustive pattern matching and the framework's Rust-specific idiom prompts requiring tests. Go's lower coverage may indicate that the framework's Go-specific prompts need refinement, or that Go's explicit error handling creates more code paths that tests do not exercise.

5.6 Code Idiomaticness

We evaluated each migration for adherence to target language idioms using an LLM-based reviewer. The reviewer examined source files against language-specific criteria: for Rust, proper use of Result/Option, ownership patterns, and pattern matching; for Java, encapsulation, naming conventions, and exception handling; for Go, error handling patterns, interface design, and struct organization. Scores were assigned on a three-tier scale: *Idiomatic* (follows conventions well, would pass code review), *Acceptable* (works but has non-idiomatic patterns), or *Non-idiomatic* (significant style issues).

All six migrations scored **Idiomatic**. The Rust migrations demonstrated proper error handling with the `?` operator, correct ownership with `Box` for recursive structures, and comprehensive pattern matching. Java migrations used sealed interfaces, immutable classes, and proper JavaDoc documentation. Go migrations followed the `if err != nil` pattern, used idiomatic `NewXxx` constructors, and maintained clean struct design.

This uniformly high idiomaticness score suggests that the quality gates (Clippy for Rust, Checkstyle for Java, `go vet` for Go) effectively enforce language conventions during migration. The autonomous feedback loop between code generation and linting appears sufficient to produce code that would pass professional code review.

5.7 Cost Observations

Across all six migrations, observed costs ranged from $6.43 to $14.11, with wall-clock durations from 32 to 60 minutes. For this 352-line source codebase, this corresponds to approximately $18 to $40 per thousand lines of source code.

The wide cost range ($6.43 to $14.11) from single runs highlights the variance in LLM-based migration. The Java module-by-module outlier ($14.11) was more than double the least expensive run ($6.43 for Go feature-by-feature). This variance makes per-KLOC cost projections unreliable without multiple runs per configuration.

All runs completed successfully, suggesting the methodology is reliable for correctness even if cost varies substantially between runs.

5.8 Scope Characterization and Limitations

The success of our methodology must be understood within its demonstrated scope. Our subject system represents a deliberately *trivial* case: 352 lines of production code, no external dependencies, low cyclomatic complexity, unidirectional module dependencies, and deterministic input-output behavior. These characteristics define the boundary conditions under which we have validated the approach.

What we have proven: LLM-based multi-agent migration can achieve 100% behavioral equivalence for small-scale, well-structured codebases across multiple target languages.

What remains unproven: Whether the methodology scales to medium or large codebases with:

- External library dependencies requiring equivalent library selection in the target language
- Cross-module circular dependencies or complex initialization ordering
- Non-deterministic behavior (threading, I/O, randomness)
- Legacy code patterns lacking clear module boundaries
- Implicit behavioral contracts not captured by simple I/O testing

The 21-case behavioral contract, while comprehensive for our subject system, represents only the most straightforward form of behavioral specification. Systems with stateful behavior, side effects, or complex error recovery may require substantially richer contracts.

6 Related Work

This work builds on research in LLM-based code generation, multi-agent systems, and automated software migration. Prior work on code generation has focused primarily on single-function or single-file generation from natural language descriptions. Our work extends this to complete multi-file codebase migration with behavioral validation.

Multi-agent LLM systems have been explored for complex reasoning tasks, but their application to software engineering tasks with tool use remains limited. Our agent architecture with specialized roles and model selection contributes to understanding of effective multi-agent designs.

Automated software migration has traditionally relied on rule-based transformation systems or statistical translation models. LLM-based approaches offer greater flexibility but require careful attention to behavioral equivalence that rule-based systems handle implicitly.

7 Conclusion

This paper provides empirical evidence that LLM-based multi-agent systems can automate cross-language code migration for small-scale, well-structured codebases. Through six migrations to Rust, Java, and Go using two strategies, we validated a four-phase methodology that successfully produces working code across structurally different target languages.

Primary finding: The methodology is language-agnostic. The same four-phase process—I/O contract generation, source analysis, sequential migration, and behavioral review—succeeded for all three target languages. Only configuration changes were required: build commands, file extensions, and idiom-specific prompts.

Coverage finding: Test coverage varied substantially by target language: Rust achieved 94–95%, Java 73–84%, and Go 65–68%. This disparity suggests that language-specific prompts influence test generation quality and warrants investigation.

Cost finding: Observed costs ranged from $6.43 to $14.11 per migration, a 2.2x variance from single runs. This variance makes cost prediction unreliable without multiple runs per configuration.

Strategy finding: Module-by-module completed faster (32–37 minutes) than feature-by-feature (55–60 minutes). Cost showed no consistent pattern by strategy. We cannot determine strategy superiority from single runs.

Our subject system, `rpn2tex`, is deliberately trivial: 352 lines of production code, no external dependencies, low complexity, and deterministic behavior. This triviality establishes that the fundamental approach works. The research question now shifts from feasibility to scalability and variance characterization.

8 Future Work

The validated methodology provides a foundation for scaling research. Key questions for future investigation include:

Variance characterization: The 2.2x cost variance ($6.43 to $14.11) and coverage variance (65–95%) observed from single runs necessitate multiple runs per configuration. We recommend 5–10 runs per configuration to establish confidence intervals for cost, duration, and coverage metrics.

Coverage improvement: The 30-percentage-point coverage gap between Rust (94–95%) and Go (65–68%) suggests that language-specific prompts significantly influence test generation. Investigating prompt modifications to improve Go and Java coverage is a priority.

Medium-complexity codebases: How does the methodology perform on systems in the 5,000–20,000 LOC range with external dependencies? At larger scale, variance characterization becomes more critical for practical deployment decisions.

Strategy selection: Module-by-module was consistently faster, but neither strategy showed consistent cost advantage. Understanding when each strategy is preferable requires controlled experiments with multiple runs.

Dependency management: Can agents automatically select equivalent libraries in the target ecosystem, or does this require human guidance?

Stateful systems: How can behavioral contracts capture systems with internal state, side effects, or non-deterministic behavior?

The success demonstrated in this paper establishes that the approach works at small scale. The challenge now is characterizing variance and determining scalability boundaries.

Acknowledgments

This research was conducted using the Claude Agent SDK with Claude 3.5 Sonnet and Claude 3.5 Haiku models. The experimental framework and analysis were developed collaboratively between human and AI researchers.

Data Availability

The migration framework, experimental data, and all generated code are available at: `https://github.com/jmf-pobox/llm-migration-research`

References

[1] Anthropic. *Claude Agent SDK Documentation*. 2024.

[2] The Rust Programming Language. *The Rust Reference*. 2024.

A Complete Migration Metrics

Table 6 presents all collected metrics for the six migrations performed in this study.

Table 6: Complete migration metrics for all configurations

	Module-by-Module			Feature-by-Feature		
Metric	Rust	Java	Go	Rust	Java	Go
Lines of Code						
Production LOC	601	553	856	513	516	479
Test LOC	1,070	1,805	3,207	1,354	1,022	1,380
Test/Prod ratio	1.8x	3.3x	3.7x	2.6x	2.0x	2.9x
Timing						
Wall clock (min)	32	37	37	60	55	56
API time (min)	66	85	66	68	69	149
Cost & Tokens						
Total cost (USD)	8.83	14.11	8.99	9.60	8.18	6.43
Output tokens	16,119	13,764	10,291	18,291	10,226	19,005
Cache read tokens	149,507	198,183	151,472	280,854	193,786	551,784
Agent Activity						
Total messages	906	1,370	1,009	779	779	853
Orchestrator turns	17	28	17	21	19	32
Migrator invocations	7	7	7	6	6	6
Reviewer invocations	7	7	7	6	2	6
Tool Invocations						
Bash	237	334	291	178	157	196
Read	123	184	119	108	119	131
Write	30	57	48	24	37	35
Edit	14	42	1	47	31	17
Quality						
Line coverage (%)	94.7	84.0	64.9	94.8	73.0	68.2
Idiomaticness	Idiomatic	Idiomatic	Idiomatic	Idiomatic	Idiomatic	Idiomatic

Notes:

- All migrations used Claude 3.5 Sonnet for code generation and Claude 3.5 Haiku for analysis/review
- Source codebase: 352 lines of Python production code + 204 lines of test code (0.6x test/prod ratio)
- Agents generated 2–4x more test code than production code, a 3–6x increase over source ratio
- Each configuration was run once; values represent single observations
- Cache read tokens indicate prompt caching efficiency (higher = more cache hits)
- Java FbF had only 2 reviewer invocations vs 6 for others, possibly due to early termination