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 demonstrates that LLM-based multi-agent systems can reliably perform automated cross-language code migration for small-scale, well-structured codebases. We present empirical validation using a 990-line Python codebase migrated to two structurally different target languages: Rust (a systems language with ownership semantics) and Java (a managed language with garbage collection). Both migrations achieved 100% behavioral equivalence with the source implementation, as verified through input-output contract validation. The Rust migration produced 1,158 lines with 97.66% test coverage at $3.74 USD; the Java migration produced 1,262 lines with 95.87% test coverage at $7.24 USD. We evaluated two migration strategies: *module-by-module* (vertical slices) and *feature-by-feature* (horizontal slices), finding both produce identical behavioral output with different code organization. Our four-phase methodology—I/O contract generation, source analysis, sequential migration, and behavioral review—proved language-agnostic, requiring only configuration changes to target different languages. These results establish that automated migration is achievable for codebases exhibiting low cyclomatic complexity, strong module boundaries, and comprehensive test coverage. The methodology's limitations at this scale—primarily the trivial nature of the subject system—motivate future research into scaling these techniques to medium-complexity codebases with cross-module dependencies and external integrations.

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 six experimental runs—five migrating Python to Rust, and one migrating Python to Java—we developed and validated a four-phase methodology that achieves 100% behavioral equivalence across multiple target languages. The successful migration to two structurally different languages (Rust's ownership model versus Java's garbage collection) demonstrates that our methodology is language-agnostic rather than target-specific.

1.1 Problem Statement

Traditional LLM-based code generation tools suffer from fundamental limitations when applied to migration tasks. 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. **Proof of concept**: We demonstrate that automated migration achieving 100% behavioral equivalence is achievable for small-scale, well-structured codebases.

2. **Multi-language validation**: We show the same four-phase methodology succeeds for both Rust and Java, 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 automated 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 990 lines of 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

Table 1 presents production code metrics across all implementations, measured using line counts excluding test code.

Table 1: Production code metrics (excluding tests)

Target	Strategy	LOC	Funcs
Python	(source)	352	25
Rust	Mod-by-mod	579	38
Rust	Feat-by-feat	549	33
Java	Mod-by-mod	368	31
Java	Feat-by-feat	389	32

Both migration strategies produced similar amounts of production code for each target language. Rust migrations expanded from 352 to approximately 560 LOC (1.6x), while Java migrations expanded to approximately 380 LOC (1.1x). The higher expansion ratio for Rust likely reflects explicit type annotations, pattern matching, and Result-based error handling.

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. The Rust compiler must accept the code without errors. The Clippy linter must report zero warnings when run in strict mode. The code formatter must not require any changes. All unit tests and documentation 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.

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 two structurally different target languages: Rust (a systems language with ownership semantics and no garbage collection) and Java (a managed language with garbage collection). 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
Rust	Mod-by-mod	32m	$6.53	937
Rust	Feat-by-feat	32m	$4.63	742
Java	Mod-by-mod	26m	$4.92	871
Java	Feat-by-feat	51m	$6.27	1026

All four migrations achieved 100% I/O contract match. The observed cost and duration differences between strategies varied by target language: for Rust, feature-by-feature was less expensive; for Java, module-by-module was less expensive.

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. Establishing statistically significant conclusions about strategy efficiency would require 5–10 runs per configuration to characterize the distribution of outcomes.

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	API	Msgs
Rust	Mod-by-mod	$6.53	55m	937
Rust	Feat-by-feat	$4.63	33m	742
Java	Mod-by-mod	$4.92	45m	871
Java	Feat-by-feat	$6.27	54m	1026

Both strategies produced identical output for all test cases regardless of target language. The observed pattern—feature-by-feature appearing more efficient for Rust while module-by-module appearing more efficient for Java—may reflect genuine differences in how each strategy interacts with target language characteristics, or may reflect run-to-run variance in non-deterministic model behavior.

We note several possible explanations for the observed differences, though none are confirmed:

- Rust's module system and cargo's incremental compilation may favor feature-by-feature's iterative approach
- Java's class-based structure may align better with module-by-module's complete-class-at-a-time approach
- The specific test cases and error recovery paths encountered in each run vary non-deterministically

Both strategies reliably produce correct output. For this trivial codebase, strategy selection does not affect correctness, only cost and duration.

5.4 Cost Observations

Across all four migrations (two targets, two strategies), observed costs ranged from $4.63 to $6.53, with durations from 26 to 51 minutes. For this 990-line source codebase, this corresponds to approximately $4.70 to $6.60 per thousand lines of source code.

These observations derive from single runs per configuration. The non-deterministic nature of LLM-based migration means individual runs may vary. To establish reliable cost estimates, multiple runs per configuration would be necessary to characterize the distribution.

We note that all runs completed successfully with 100% I/O contract match, suggesting the methodology is reliable for correctness even if cost varies between runs.

5.5 Scope Characterization and Limitations

The success of our methodology must be understood within its demonstrated scope. Our subject system represents a deliberately *trivial* case: under 1,000 lines, 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 reliably automate cross-language code migration for small-scale, well-structured codebases. Through successful migrations to both Rust and Java, we have validated a four-phase methodology that achieves 100% behavioral equivalence with the source implementation.

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 both Rust (a systems language with ownership semantics) and Java (a managed language with garbage collection). Only configuration changes were required: build commands, file extensions, and idiom-specific prompts. This validates that the approach is not target-specific.

Secondary finding: Behavioral contracts are essential. Without explicit I/O contract validation, migrations may produce functionally correct implementations that differ semantically from the source. Early experiments showed 19% behavioral discrepancies despite passing all compilation, linting, and testing gates, demonstrating that traditional quality metrics are insufficient for faithful migration.

Methodological finding: Smaller, focused agent contexts outperform large comprehensive prompts. Multi-phase orchestration with focused specification documents achieved results four times faster and 25% cheaper than embedding source code directly in prompts.

Our subject system, `rpn2tex`, is deliberately trivial: under 1,000 lines, no external dependencies, low complexity, and deterministic behavior. This triviality is a feature, not a limitation. By demonstrating success on an ideal case, we establish that the fundamental approach works. The research question now shifts from feasibility to scalability.

8 Future Work

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

Medium-complexity codebases: How does the methodology perform on systems in the 5,000–20,000 LOC range with external dependencies? Our next target is `txt2tex`, a 9,300 LOC Python codebase with average cyclomatic complexity of 6.7 and maximum complexity of 40. At this scale, the relative performance of feature-by-feature versus module-by-module strategies remains an open question.

Strategy selection: When should feature-by-feature be preferred over module-by-module? Our observations showed different patterns by target language, but with only one run per configuration, we cannot determine whether these reflect systematic differences or run-to-run variance. Multiple runs per configuration would be needed to identify reliable strategy selection criteria.

Dependency management: Can agents automatically select equivalent libraries in the target ecosystem, or does this require human guidance? How should dependency version constraints be handled?

Stateful systems: How can behavioral contracts capture systems with internal state, side effects, or non-deterministic behavior? What contract languages are expressive enough for complex behavioral specifications?

Incremental migration: For large codebases, can migration proceed incrementally with interoperability between migrated and unmigrated modules? What FFI or serialization strategies enable gradual transition?

Cost scaling: Observed costs ranged from $4.70 to $6.60 per KLOC across four migrations of a single trivial system. How do costs scale with complexity? Are there complexity thresholds beyond which automated migration becomes cost-prohibitive? Characterizing the cost distribution would require additional runs.

The success demonstrated in this paper represents a necessary but not sufficient condition for practical automated migration. We have proven that the approach works at small scale with two different strategies; the challenge now is determining how far it can scale.

Acknowledgments

This research was conducted using the Claude Agent SDK with Claude Opus 4.5 and Claude 3.5 Haiku models. The experimental framework and analysis were developed collaboratively between human and AI researchers.

References

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

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

A Input-Output Contract

The behavioral contract comprised 21 test cases across eight categories: basic binary operations (4 tests), unsupported operators (3 tests), operator precedence (3 tests), associativity (2 tests), addition chains (1 test), mixed operations (4 tests), floating-point numbers (2 tests), and complex expressions (2 tests).

Representative examples include: input `5 3 +` producing output `$5 + 3$`; input `5 3 + 2 *` producing output `$( 5 + 3 ) \times 2$`; and input `2 3 ^` producing an error for unsupported operator.

B Migration Framework Architecture

This appendix describes the multi-agent framework used to perform automated migrations. The framework is implemented using the Claude Agent SDK and consists of four specialized agents coordinated by an orchestrator.

B.1 Agent Roles

The framework employs four distinct agent types, each with specific responsibilities and tool access:

1. I/O Contract Agent (Phase 0)

- *Purpose*: Generate behavioral specification by executing source implementation
- *Model*: Claude 3.5 Haiku (fast, low-cost)
- *Tools*: `Bash`, `Read`
- *Output*: Structured I/O contract mapping inputs to expected outputs

2. Analyst Agent (Phase 1)

- *Purpose*: Read all source files and produce migration specification
- *Model*: Claude 3.5 Haiku
- *Tools*: `Read`, `Glob`, `Grep` (read-only access)
- *Output*: Comprehensive specification document including I/O contract

3. Migrator Agent (Phase 2)

- *Purpose*: Convert individual modules to target language
- *Model*: Claude 3.5 Sonnet (more capable, higher cost)
- *Tools*: `Read`, `Write`, `Edit`, `Bash`, `Glob`, `Grep` (full access)
- *Output*: Target language source files passing quality gates

4. Reviewer Agent (Phase 3)

- *Purpose*: Validate migrated code against specification and I/O contract
- *Model*: Claude 3.5 Haiku
- *Tools*: `Read`, `Glob`, `Grep`, `Bash`
- *Output*: Review report with pass/fail verdict

B.2 Orchestration Flow

The main orchestrator coordinates agent invocations following a strict phase order:

```
Orchestrator
    |
    +--[Phase 0]---> I/O Contract Agent
    |                    |
    |               (executes source on test inputs)
    |                    |
    |               I/O Contract Document
    |                    |
    +--[Phase 1]---> Analyst Agent
    |                    |
    |               (reads all source files)
    |                    |
    |               Migration Specification
    |               (includes I/O contract)
    |                    |
    +--[Phase 2]---> Migrator Agent (module 1)
    |                    |
    |               <-- Quality Gate Loop -->
    |                    |
    +--[Phase 3]---> Reviewer Agent (module 1)
    |                    |
    +--[Phase 2]---> Migrator Agent (module 2)
    |                    ...
    |               (repeat for each module)
    |                    |
    +--[Complete]
```

B.3 Feedback Loops

The framework implements two levels of feedback loops to ensure correctness:

Inner Loop: Quality Gates

During Phase 2, each migrator agent operates in a feedback loop with the build system:

```
Migrator writes code
        |
        v
    [Build/Compile]---fail---> Migrator reads errors
        |                           |
       pass                    fixes code
        |                           |
        v                           v
    [Lint/Format]---fail---> Migrator reads warnings
        |                           |
       pass                    fixes code
        |                           |
        v                           |
    [Run Tests]---fail--------> Migrator reads failures
        |                           |
       pass                    fixes code
        |                           |
        v                            |
    [I/O Contract]---fail----> Migrator adjusts logic
        |
       pass
        |
        v
    Module Complete
```

For Rust migrations, quality gates are: `cargo check`, `cargo clippy -- -D warnings`, `cargo fmt`, and `cargo test`. For Java: `./gradlew compileJava`, `./gradlew checkstyleMain`, and `./gradlew test`.

Outer Loop: Review and Revision

After migration, the reviewer agent validates against the specification. If issues are found, the orchestrator can re-invoke the migrator with reviewer feedback:

```
Migrator completes module
        |
        v
    Reviewer validates
        |
    +---pass---> Next module
    |
   fail
    |
    v
Orchestrator passes feedback to Migrator
        |
        v
    Migrator revises
        |
        v
    (re-enter quality gate loop)
```

B.4 Language-Agnostic Design

The framework abstracts language-specific details into a `LanguageTarget` interface:

```
class LanguageTarget(ABC):
    name: str              # "rust", "java"
    file_extension: str    # ".rs", ".java"

    def get_quality_gates() -> list[str]
    def get_migrator_idioms() -> str
    def get_reviewer_checks() -> str
    def get_file_mapping(source_file) -> str
```

Adding a new target language requires implementing this interface with language-specific build commands, idiom requirements, and file naming conventions. The agent prompts, orchestration logic, and feedback loops remain unchanged.

B.5 Cost Optimization

The framework optimizes API costs through model selection:

- Expensive models (Sonnet) only for code generation requiring creativity
- Cheaper models (Haiku) for analysis and validation tasks
- Specification documents reduce redundant source file reads
- Sequential module migration limits context size per invocation