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 18 migrations—three runs each for 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 with 100% behavioral equivalence. Averaging across runs, costs ranged from $6.32 to $11.93 USD per migration; test coverage ranged from 67% to 95% depending on target language. Rust achieved the highest coverage (95%), followed by Java (80–92%) and Go (67–71%). Module-by-module completed faster for Rust (32 min vs 51 min) but showed no consistent speed advantage across languages. Feature-by-feature was cheaper for Java and Go. The four-phase methodology—I/O contract generation, source analysis, sequential migration, and behavioral review—proved language-agnostic. These results establish a baseline demonstrating that automated migration is achievable for small-scale, well-structured codebases. Future research will determine whether this approach scales to medium-sized systems.

1 Introduction

Cross-language code migration represents a significant challenge in software engineering. Organizations maintaining legacy codebases—whether PHP monoliths, aging Java systems, or COBOL backends—frequently need to migrate to modern languages for performance optimization, developer availability, ecosystem alignment, or architectural modernization. Traditional approaches require substantial manual effort, with developers reading source code, understanding its semantics, and rewriting equivalent implementations in the target language. For large legacy systems, this effort can span years and consume significant engineering resources.

Recent advances in Large Language Models have enabled automated code generation at unprecedented quality levels [2]. 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 18 experimental runs—three per configuration—migrating Python to Rust, Java, and Go, we developed and validated a four-phase methodology that achieves 100% behavioral equivalence across all runs and 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 (averaged across 3 runs)

Target	Strategy	Prod	Test	Ratio
Python	(source)	352	204	0.6x
Rust	MbM	617	1,665	2.7x
Rust	FbF	462	1,362	2.9x
Java	MbM	542	2,265	4.2x
Java	FbF	404	948	2.3x
Go	MbM	575	2,608	4.5x
Go	FbF	405	1,320	3.3x

A notable finding: while the source Python had a 0.6x test-to-production ratio (204 test LOC for 352 prod LOC), agents generated 2–5x more test code than production code in the target languages. The test-to-production ratio ranged from 2.3x (Java FbF) to 4.5x (Go MbM)—a 4–7x increase in relative test investment 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 [1], 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). Each configuration (target language × strategy) was executed three times to characterize run-to-run variance. Results are reported as mean ± standard deviation where applicable.

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 (mean, n=3)

Target	Strategy	Dur. (min)	Cost	Msgs	Cov.
Rust	MbM	32	$8.62	918	95%
Rust	FbF	51	$8.24	856	95%
Java	MbM	45	$11.93	1177	92%
Java	FbF	47	$7.96	938	80%
Go	MbM	49	$8.42	1018	71%
Go	FbF	44	$6.32	894	67%

All 18 migrations completed successfully and passed quality gates. Module-by-module was faster for Rust (32 min vs 51 min) but showed no consistent speed advantage for Java or Go. Feature-by-feature was consistently cheaper for Java ($7.96 vs $11.93) and Go ($6.32 vs $8.42), while costs were comparable for Rust. Test coverage varied by target language: Rust achieved 95%, Java 80–92%, and Go 67–71%.

With three runs per configuration, we observed substantial run-to-run variance in duration and cost. See Appendix for detailed variance characterization.

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 [11]. 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 [8]. 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 [13]. 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 (mean, n=3)

Target	Strategy	Cost	Dur. (min)	Msgs
Rust	MbM	$8.62	32	918
Rust	FbF	$8.24	51	856
Java	MbM	$11.93	45	1177
Java	FbF	$7.96	47	938
Go	MbM	$8.42	49	1018
Go	FbF	$6.32	44	894

Both strategies produced correct output for all 18 runs. Module-by-module was faster for Rust (32 vs 51 min) but showed no speed advantage for Java or Go. Feature-by-feature produced fewer messages and was cheaper for Java ($7.96 vs $11.93) and Go ($6.32 vs $8.42).

Java module-by-module had the highest cost ($11.93) and message count (1177), suggesting more error recovery cycles with this combination. With three runs per configuration, we conclude that strategy effects are language-dependent: module-by-module is faster for Rust, while feature-by-feature is cheaper for Java and Go.

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.

Module-by-module strategies used more Bash invocations (241–300) than feature-by-feature (185–200), consistent with more build/test cycles per module. Feature-by-feature migrations showed higher Edit-to-Write ratios (Rust FbF: 49 Edit vs 25 Write;

Table 4: Tool invocations by target and strategy (averaged across 3 runs)

Target	Strategy	Bash	Read	Write	Edit
Rust	MbM	241	123	29	19
Rust	FbF	185	123	25	49
Java	MbM	300	153	46	24
Java	FbF	200	142	34	33
Go	MbM	300	112	39	12
Go	FbF	192	141	29	29

Go FbF: 29 Edit vs 29 Write), indicating more iterative file refinement. Module-by-module favored complete file writes (Go MbM: 12 Edit vs 39 Write), suggesting a write-then-verify approach.

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 (mean, n=3)

Target	Strategy	Line Coverage
Rust	MbM	95%
Rust	FbF	95%
Java	MbM	92%
Java	FbF	80%
Go	MbM	71%
Go	FbF	67%

Rust migrations achieved the highest coverage (95%) regardless of strategy, measured using cargo-llvm-cov. Java coverage varied by strategy: module-by-module achieved 92% while feature-by-feature reached 80%, measured via JaCoCo. Go achieved the lowest coverage (67–71%), measured using Go's built-in coverage tooling.

The coverage disparity across languages is consistent across all 18 runs. Rust's high coverage may reflect the language's emphasis on exhaustive pattern matching and the framework's Rust-specific idiom prompts. Coverage variance was lowest for Rust and highest for Go (see Appendix).

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 18 migrations, mean costs per configuration ranged from $6.32 (Go FbF) to $11.93 (Java MbM). For this 352-line source codebase, this corresponds to approximately $18 to $34 per thousand lines of source code.

Feature-by-feature strategies were consistently cheaper than module-by-module for Java (34% cheaper) and Go (25% cheaper), while costs were comparable for Rust. The 1.9x ratio between highest and lowest mean costs ($11.93 vs $6.32) reflects meaningful differences between configurations rather than run-to-run noise.

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.

Code generation benchmarks. Prior work on LLM code generation has focused primarily on single-function or single-file generation. HumanEval [2] evaluates function-level synthesis from docstrings, while AlphaCode [7] generates solutions to competitive programming problems. More recently, SWE-bench [5] introduced repository-level evaluation using real GitHub issues, representing a shift toward more realistic software engineering tasks. Our work extends this trajectory to complete cross-language migration with behavioral validation.

Multi-agent software engineering. Concurrent with our work, multi-agent LLM systems have been applied to software development. MetaGPT [4] encodes software development SOPs into multi-agent workflows with specialized roles (product manager, architect, engineer). ChatDev [10] implements waterfall-style development through communicative agents. These systems generate new software from specifications rather than migrating existing codebases, but share architectural principles with our approach: specialized agent roles and iterative refinement through quality gates.

LLM-based code translation. TransAGENT [14] presents a multi-agent system for code translation with four specialized agents: translator, syntax fixer, code aligner, and semantic error fixer. The system uses execution alignment to localize errors in translated code, enabling targeted fixes. Our work differs in scope and validation approach: TransAGENT targets function-level translation with execution-based debugging, while we demonstrate end-to-end codebase migration validated against I/O contracts derived from source behavior. The I/O contract approach provides an oracle for behavioral equivalence without requiring execution alignment infrastructure.

Industrial-scale migration. Ziftci et al. [15] report on LLM-assisted migrations at Google, achieving 74% LLM-generated code changes across 39 migrations in their monorepo. Their system runs autonomously but addresses same-language migrations (API updates, dependency upgrades) rather than cross-language translation, and relies on existing test suites for validation. Our work complements this by demonstrating cross-language migration where target-language tests do not yet exist, requiring behavioral contracts generated from source execution.

Rule-based transformation. Traditional transpilers rely on rule-based systems such as TXL [3], Stratego/XT [12], and Rascal [6]. These require manually specified transformation rules, which be-

come impractical for large API surfaces and cannot adapt to idiomatic target-language patterns. Recent work [9] shows LLM-based translation outperforms rule-based approaches in accuracy while producing more concise, idiomatic output.

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 18 migrations—three runs each for two strategies across three target languages—we validated a four-phase methodology that reliably produces working code across structurally different target languages.

Primary finding: The methodology is language-agnostic and reliable. All 18 migrations achieved 100% behavioral equivalence. The same four-phase process—I/O contract generation, source analysis, sequential migration, and behavioral review—succeeded for Rust, Java, and Go with only configuration changes.

Coverage finding: Test coverage varied by target language: Rust achieved 95%, Java ranged from 80–92%, and Go achieved 67–71%. Rust's consistency suggests its toolchain and language features support more predictable test generation.

Cost finding: Mean costs ranged from $6.32 to $11.93 per migration (1.9x range). Feature-by-feature was consistently cheaper for Java (34%) and Go (25%).

Strategy finding: Strategy effects were language-dependent. Module-by-module was faster for Rust (32 vs 51 min) but showed no speed advantage for Java or Go. Feature-by-feature produced fewer messages and lower costs for Java and Go.

Our subject system, `rpn2tex`, is deliberately trivial: 352 lines of production code, no external dependencies, and deterministic behavior. This establishes a baseline demonstrating that automated migration works at small scale. Future research will determine whether this approach scales to medium-sized codebases (5,000–20,000 LOC) with external dependencies.

8 Future Work

The validated methodology and variance characterization from 18 runs provide a foundation for scaling research toward practical legacy modernization. Key questions for future investigation include:

Incremental monolith decomposition: The most promising path to migrating large legacy codebases (PHP, COBOL, aging Java) may not be whole-system translation but incremental extraction. Can agents identify and extract small, well-designed modules suitable for automated migration—analogous to "extract method" refactoring at the module level? Each extracted module would meet the characteristics that made our subject system amenable to automation: clear boundaries, low complexity, and minimal dependencies. This approach would enable organizations to progressively decompose monolithic systems into modern microservices, migrating one bounded context at a time while maintaining system integrity.

Medium-complexity codebases: The primary scaling question is whether this methodology extends to systems in the 5,000–20,000 LOC range with external dependencies. Google's experience [15] suggests LLM-based migration can work at scale for same-language transformations; cross-language migration at scale remains unproven. At larger scale, cost and duration variance become more critical for practical deployment decisions.

Coverage improvement: The 28-percentage-point coverage gap between Rust (95%) and Go (67–71%) persisted across all runs. Investigating prompt modifications to improve Go and Java coverage is a priority for production readiness.

Dependency management: Can agents automatically select equivalent libraries in the target ecosystem, or does this require human guidance? A PHP application using Laravel would need equivalent frameworks in the target language—this library mapping problem is a prerequisite for medium-scale migration.

Stateful systems: How can behavioral contracts capture systems with internal state, side effects, or non-deterministic behavior? Our current I/O contract approach assumes pure, deterministic functions. Real legacy systems often have database interactions, session state, and external API calls that re-

quire richer behavioral specifications.

Strategy selection guidelines: Strategy effects were language-dependent: module-by-module was faster for Rust but feature-by-feature was cheaper for Java and Go. Developing heuristics for strategy selection based on target language and codebase characteristics would improve migration efficiency.

This baseline establishes that automated migration works reliably at small scale with 100% behavioral equivalence across 18 runs. The challenge now is extending these techniques to enable practical legacy modernization—helping organizations escape the maintenance burden of aging codebases by providing a reliable, cost-effective path to modern languages and architectures.

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`

A Complete Migration Metrics

Table 6 presents averaged metrics across all 18 migrations (3 runs per configuration).

Table 6: Complete migration metrics for all configurations (mean of 3 runs)

	Module-by-Module			Feature-by-Feature		
Metric	Rust	Java	Go	Rust	Java	Go
Lines of Code						
Production LOC	617	542	575	462	404	405
Test LOC	1,665	2,265	2,608	1,362	948	1,320
Test/Prod ratio	2.7x	4.2x	4.5x	2.9x	2.3x	3.3x
Timing (mean ± std)						
Wall clock (min)	32 ± 6	45 ± 15	49 ± 18	51 ± 8	47 ± 10	44 ± 12
Cost (mean ± std)						
Total cost (USD)	8.62 ± 0.64	11.93 ± 1.93	8.42 ± 0.50	8.24 ± 1.18	7.96 ± 0.31	6.32 ± 0.11
Agent Activity (mean)						
Total messages	918	1,177	1,018	856	938	894
Tool Invocations (mean)						
Bash	241	300	300	185	200	192
Read	123	153	112	123	142	141
Write	29	46	39	25	34	29
Edit	19	24	12	49	33	29
Quality (mean ± std)						
Line coverage (%)	95.0 ± 2.3	91.7 ± 4.3	71.1 ± 7.5	94.9 ± 0.2	80.3 ± 6.6	66.7 ± 2.3
I/O match rate	100%	100%	100%	100%	100%	100%
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 2–5x more test code than production code, a 4–7x increase over source ratio
- Each configuration was run 3 times; values represent means with standard deviations where applicable
- All 18 runs achieved 100% behavioral equivalence on the 21-case I/O contract

References

[1] Anthropic. Claude agent SDK documentation. `https://github.com/anthropics/claude-code`, 2024.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] James R. Cordy. TXL—a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2006.

[4] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024.

[5] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

[6] Paul Klint, Tijs van der Storm, and Jurgen Vinju. RASCAL: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009.

[7] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[8] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[9] Zhen Pan, Shuai Lu, and Michael R. Lyu. Exploring and unleashing the power of large language models in automated code translation. *Proceedings of the ACM on Software Engineering*, 1(FSE), 2024.

[10] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2024.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

[12] Eelco Visser. Program transformation with Stratego/XT. In *Domain-Specific Program Generation*, pages 216–238. Springer, 2004.

[13] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022.

[14] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. TransAGENT: An LLM-based multi-agent system for code translation. *arXiv preprint arXiv:2409.19894*, 2024.

[15] Celal Ziftci, Diego Cavalcanti, Clémentine Fourrier, and Martin Schaef. Migrating code at scale with LLMs at Google. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. ACM, 2025.