Transactions on Computer Science and Intelligent Systems Research
ISSN: 2960-1800, eISSN: 2960-2238 | Volume 13
AMMSAC 2026

ArchMigrate-Swift: A Compiler-in-the-Loop Retrieval-Augmented Multi-Agent Framework for Objective-C-to-Swift Migration

Wenbin Shang [1,*], Jie-Si Yang [2], Guanyu Ding [3], Shiyu Yang [4]

[1] University of Glasgow, G12 8QQ Glasgow, U.K.

[2] The University of Utah, Salt Lake City, UT 84112, USA

[3] New York University, New York, NY 10012, USA

[4] University of California, Los Angeles, Los Angeles, CA 90095, USA

* Corresponding Author: shang.wenbin@ieee.org

Abstract. Migrating Objective-C systems to Swift is not a token-level translation problem: API-name import rules, Foundation bridging, value semantics, optionality, collection behavior, and project-specific contracts interact with compiler constraints. This paper presents ArchMigrate-Swift, a compiler-in-the-loop, retrieval-augmented multi-agent framework that decomposes migration into schema retrieval, structural intent analysis, candidate generation, diagnostic repair, executable criticism, and evidence-based arbitration. The architecture is language-model agnostic; the released artifact instantiates every role deterministically so that the reported results do not depend on proprietary services or stochastic model drift. Because no established paired Objective-C–Swift benchmark with executable tests was identified, we introduce AMSwiftBench, a fixed-seed controlled benchmark containing 48 retrieval exemplars and 96 held-out Objective-C tasks spanning 12 migration families and eight lexical/structural difficulty modes. All generated Swift candidates are checked by Swift 6.2.1 and executed against public and hidden tests. In the controlled closed-schema setting, retrieval raises family identification from 94.8% to 100%; compiler repair raises compilation from 79.2% to 100%; and public-test-only arbitration raises end-to-end pass rate from 22.9% to 100%, at 190.3 ms amortized wall time per task. Exact McNemar analysis shows that the full system improves 63 of 96 tasks over retrieval plus compiler repair without regressions ($p=2.17\times10^{-19}$). These results establish the value of separating syntactic validity from semantic selection, while explicitly not claiming industrial end-to-end migration performance.

Keywords: Program migration; Objective-C; Swift; Compiler feedback; Retrieval-augmented generation; Multi-agent systems; Executable evaluation.

1. Introduction

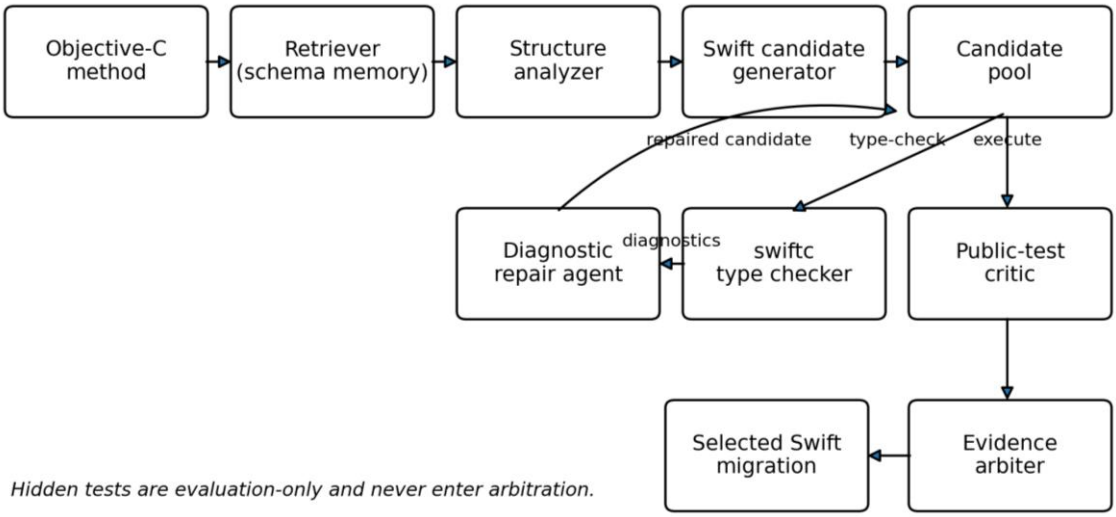

Figure 1. ArchMigrate-Swift workflow. Hidden tests are isolated from candidate selection.

Apple supports mixed-language projects and recommends incremental Objective-C-to-Swift migration rather than an all-at-once rewrite [1], [2]. Yet a source method that appears locally mechanical may depend on Clang-importer name translation, Foundation type bridging, nullability, Objective-C runtime behavior, or a contract encoded only in tests [3-5]. Empirical studies show that Swift adoption and language mixing are observable at ecosystem scale [6], [7], while practical converters and prompt toolkits remain explicitly assistive rather than guarantees of behavior preservation [8], [9]. The central difficulty is therefore architectural: a migration system must gather relevant precedents, reason about intent, satisfy a changing target compiler, and reject compiling but semantically wrong candidates.

Neural transcompilers demonstrate that multilingual representations can learn broad code mappings [10-20]. Retrieval-augmented generation (RAG) provides non-parametric memory [21], and iterative feedback methods show that external evidence can improve generated artifacts [22], [23]. Multi-agent software systems further separate planning, coding, testing, and review responsibilities [24-27]. However, these strands do not by themselves resolve three migration-specific failure modes. First, retrieval can identify the right operation while generation still emits stale or fictitious Swift constructs. Second, compiler success proves type and syntax compatibility, not behavioral equivalence. Third, evaluating only textual similarity can reward translations that look plausible but violate contracts; execution-based code benchmarks motivate stronger evaluation [15], [29-31].

ArchMigrate-Swift addresses these gaps with a staged evidence pipeline. A schema memory retrieves migration precedents; a structure analyzer re-ranks them using Objective-C cues; a generator proposes Swift candidates; the actual Swift compiler supplies diagnostics; a repair role applies diagnostic-grounded edits; a critic executes public tests; and an arbiter selects using only deployable evidence. The design is compatible with LLM-based roles, but this paper evaluates a deterministic instantiation. That choice is deliberate: the experiment measures architectural contribution without confounding model updates, hidden training data, temperature, or API availability.

The contributions are threefold. (1) We formulate Objective-C-to-Swift migration as constrained evidence aggregation rather than one-shot translation, with compiler diagnostics and public tests assigned distinct roles. (2) We release AMSwiftBench, a transparent fixed-seed generator with 48 retrieval exemplars and 96 held-out tasks across documented migration idioms. (3) We report an ablation with actual Swift compilation and execution, per-task logs, bootstrap intervals, exact paired tests, and a reproducibility package. The 100% result is confined to a known-family, unseen-variant benchmark and is not an estimate for UIKit-, AppKit-, macro-, runtime-, or repository-scale migration.

2. Background And Related Work

2.1. Objective-C–Swift Interoperability

Objective-C and Swift can coexist in one target through generated interfaces and bridging headers [2]. The imported Swift surface may differ substantially from source spelling: SE-0005 specifies linguistic and type-aware Objective-C name translation [4], and Apple documents annotations that rename, reorder, or reshape APIs for Swift [3]. Foundation reference types can bridge to native Swift value types, making collection and nullability choices semantic rather than cosmetic. Apple therefore recommends migration in tested increments [1]. These rules motivate a migration memory organized by semantic schema—such as “dictionary lookup with default” or “nullable division”—instead of exact token overlap.

2.2. Code Translation and Executable Evaluation

TransCoder learns unsupervised translation among C++, Java, and Python [10]; its follow-up uses generated unit tests to filter translations [11]. Compiler intermediate representations have also been used to inject language-independent structure [12]. XLCoST and CodeNet provide broad multilingual resources [13], [14], whereas xCodeEval emphasizes executable, multilingual evaluation [15]. Pre-

trained code models—CodeBERT, GraphCodeBERT, CodeT5, PLBART, and DOBF—show that identifiers, data flow, denoising, and structure improve downstream code tasks [16-20]. None of these datasets contains a verified paired Objective-C-to-modern-Swift migration suite with target-compiler tests, which motivates a controlled benchmark rather than relabeling unrelated language pairs.

2.3. Retrieval, Agents, and Repair

RAG combines parametric generation with retrieved evidence [21]. Self-Refine and Reflexion demonstrate iterative use of feedback without necessarily changing model weights [22], [23]. AutoGen, MetaGPT, and ChatDev organize specialized conversational roles [24-26], while AgentCoder separates programming, test design, and test execution [27]. RepairAgent lets an LLM invoke tools and validate patches autonomously [28]. ArchMigrate-Swift adopts role specialization but constrains interaction through typed artifacts: ranked schemas, candidate code, compiler diagnostics, test vectors, and a deterministic selection tuple. This reduces unstructured cross-agent discussion and makes every decision auditable. Mo et al. further present an elegant dual-memory agent that combines short-term code retrieval, long-term strategy learning, and execution-driven prompt adaptation for testing and bug localization [33]. Their results reinforce the value of separating immediate context from reusable cross-task experience.

3. Problem Formulation

Given an Objective-C method x, repository evidence E, target compiler C, public tests T_p, and hidden evaluation tests T_h, the goal is to produce Swift program y that compiles under C and preserves the method contract. T_h is unavailable to the system and is used only for evaluation. The released benchmark treats each task as a pure method so that correctness can be determined by execution. A candidate is end-to-end correct only when it passes compilation, all public tests, and all hidden tests.

$$y^* = \arg\max_{\gamma} \text{Evidence}(y \mid x, E, C, T_p), \quad \text{subject to } C(y)=\text{pass}.$$

This formulation separates four measurable properties: family identification, compilation, public-test satisfaction, and held-out behavior. It also distinguishes repair from search. A compiler-guided edit can transform an invalid candidate into a valid one without changing an incorrect algorithm; conversely, candidate arbitration can select a semantically correct program only when alternatives exist. The experimental configurations isolate these effects.

4. Archmigrate-Swift

4.1. Schema Memory and Hybrid Retrieval

The memory stores Objective-C examples labeled with a migration family, a difficulty mode, and executable tests. For a query, word 1–2 gram TF–IDF and character 3–5 gram TF–IDF similarities are combined. Character features tolerate identifier variations, while word features preserve operation-level cues.

$$s_r(q,e) = 0.55\cos(vw(q), vw(e)) + 0.45\cos(vc(q), vc(e)).$$

The retriever returns at most one exemplar per family. A structure analyzer then extracts binary cues—ternary selection, bound checks, accumulation, increment, division, character indexing, substring APIs, dictionary access, set construction, nil guards, multiplicative recurrence, and Fibonacci state. Each family has a sparse prototype. The re-ranked score is $0.72s_r+0.38s_s$ minus mutually exclusive penalties. These weights are fixed design constants, not tuned on test outcomes.

4.2. Candidate Generation

The generator maps the highest-ranked family to a native Swift signature and body. In a learned deployment this role could be a code LLM conditioned on retrieved examples. For reproducibility, the artifact uses schemas and deterministically assigns one of five initial variants from a SHA-256 hash of the task identifier: correct, semantic error, legacy type, invalid member, or invalid null sentinel. The full system adds a second alternative—normally the correct schema—to create a minimal candidate pool. This controlled injection supports causal ablation: every configuration sees the same task-level initial variant.

4.3. Compiler-in-the-Loop Repair

Candidates are wrapped in isolated Swift enums and type-checked by swiftc. To reduce process startup cost, up to 48 candidates are placed in one file; line ranges map diagnostics back to candidate identifiers. The repair role performs only edits justified by observed diagnostics: LegacyInteger→Int, length→count, includes→contains, removal of an unresolved legacyMember, and LegacyNull→nil. Failed candidates receive at most one repair round and are type-checked again. This rule-based realization is intentionally narrow; it prevents a reported repair gain from being attributed to an opaque prompt or unobserved manual intervention. Zhao et al. likewise show that a RAG-enhanced generation pipeline benefits from a dedicated pre-validation agent that detects deployment constraints before execution [32]. Their design provides a strong precedent for treating validation as a first-class, closed-loop generation stage.

4.4. Executable Critic and Evidence Arbiter

Compiling code is insufficient because a wrong maximum, off-by-one recurrence, dropped array element, or incorrect default can type-check. Therefore, every compiling candidate is executed against four public tests. The arbiter selects the lexicographic maximum of (public-pass, compile-pass, retrieval-score, negative public failures, negative candidate index). Hidden outcomes are never included. Public and hidden calls are emitted into one executable solely to amortize compilation; output markers keep the splits distinct, and selection occurs before hidden results are read by the evaluation layer.

$$a(y) = (1[T_p(y)=\text{pass}],\ 1[C(y)=\text{pass}],\ s(y),\ -\text{fail}_p(y),\ -\text{index}(y)).$$

4.5. Agent Contracts and Failure Containment

Each role consumes and emits a small typed record. Retrieval cannot edit code; repair cannot invent a new family; the critic reports observations rather than altering candidates; and the arbiter cannot access hidden tests. This division limits cascading hallucination in an LLM-backed implementation and makes the deterministic artifact a reference semantics for future learned agents. If the family is wrong, execution is marked failed rather than invoking a test harness with an incompatible signature.

4.6. Termination, Complexity, and Audit Trail

The released loop terminates because retrieval is finite, the generator emits at most two candidates per task, and repair is capped at one round. With n tasks, k retrieved families, m candidates, and compiler batch size b, retrieval cost is dominated by sparse cosine similarity, while compiler invocations are O(ceil(nm/b)). Every candidate record stores the exemplar identifier, retrieval score, initial and selected variant, repair actions, diagnostic class, public/hidden failure counts, and amortized timing. This audit trail permits independent recomputation of every aggregate table.

Table 1. Released Archmigrate-Swift Inference

Step	Operation
1	Retrieve top-k families; structurally re-rank.
2	Generate one draft; Full adds one alternative.
3	Batch type-check with swiftc; map line diagnostics.
4	Apply one diagnostic-grounded repair; recheck.
5	Execute public tests; rank candidates without hidden data.
6	Return selected Swift code and evidence record.

5. Amswiftbench And Experimental Method

5.1. Benchmark Construction

No public paired dataset found during this study simultaneously offered Objective-C inputs, modern Swift targets, and executable tests. We therefore generated AMSwiftBench from migration patterns documented by Apple and commonly represented in source converters, while avoiding copied repository code. The fixed seed is 20260730. The training split contains four exemplars per family (48 total); the test split contains eight held-out source variants per family (96 total). Every task has four public and six hidden test cases generated from an independent reference function.

Table 2. Amswiftbench coverage

Dimension	Coverage
Families	12: numeric, collection, string, map, optional, recurrence
Difficulties	8: lexical, alias, structural, distractor, nested, bridge, compact, verbose
Splits	48 exemplars; 96 held-out tasks
Tests	4 public + 6 hidden per task

The difficulty modes change identifiers, syntax shape, verbosity, Foundation types, distractor cues, or control-flow nesting while retaining the reference behavior. The benchmark is closed-schema: every test operation belongs to a family represented in memory, but its source variant is held out. This setting measures architectural robustness to surface and structural variation, not discovery of arbitrary application behavior.

5.2. Research Questions

RQ1 asks whether retrieval and structural cues identify the correct migration schema under held-out surface variation. RQ2 asks whether compiler diagnostics recover syntactic validity without semantic oracle access. RQ3 asks whether public-test arbitration selects behaviorally correct alternatives and generalizes to hidden tests. RQ4 measures the latency cost of each evidence source. The ablation is ordered so that each configuration adds one capability while preserving the same deterministic initial draft per task.

5.3. Configurations and Metrics

Table 3. Ablation configurations

Config.	Retrieval	Repair	Selection
Direct	Keywords	No	1 draft
RAG	TF–IDF	No	1 draft
RAG+Comp.	TF–IDF	Yes	1 draft
Full	Hybrid	Yes	2 + public tests

We report family accuracy, compile rate, public pass rate, held-out pass rate, end-to-end pass rate (public and hidden both pass), and amortized wall/compiler time. Functional execution is preferred to CodeBLEU because semantically equivalent implementations may differ lexically [31]. For uncertainty, 10,000 task-level bootstrap resamples provide percentile 95% intervals for held-out pass rate. Exact two-sided McNemar tests compare paired task outcomes; no independence across configurations is assumed.

5.4. Implementation and Reproducibility

Experiments ran locally with Python 3.13.5, scikit-learn TF–IDF retrieval, and Swift 6.2.1 targeting x86_64 Linux. The evaluator invokes swiftc -typecheck, compiles executable harnesses, and records per-task outcomes in CSV. Timing is amortized over batch compilation and includes framework work. The package contains the generator, source/target schemas, evaluator, raw CSV, environment capture, bootstrap results, and exact McNemar counts. No network API or manual correction is used during evaluation.

6. Results

6.1. Main Ablation

Table 4. Results On 96 Held-Out Tasks

Config.	Fam.	Comp.	Public	Hidden	E2E	ms/task
Direct	94.8	79.2	20.8	29.2	20.8	35.3
RAG	100.0	79.2	22.9	31.3	22.9	39.3
RAG+Comp.	100.0	100.0	22.9	34.4	22.9	56.9
Full	100.0	100.0	100.0	100.0	100.0	190.3

Table 4 and Fig. 2 show a staged result. Retrieval eliminates five family-identification errors, raising accuracy from 94.8% to 100%, but yields only two additional end-to-end successes. Compiler repair fixes all 20 selected compile failures and raises compile rate from 79.2% to 100%; it does not improve end-to-end success because the repaired drafts often retain their deliberately injected semantic error. The full system succeeds on all 96 tasks because its candidate pool contains a behaviorally correct alternative and the public tests distinguish it from the preferred faulty draft.

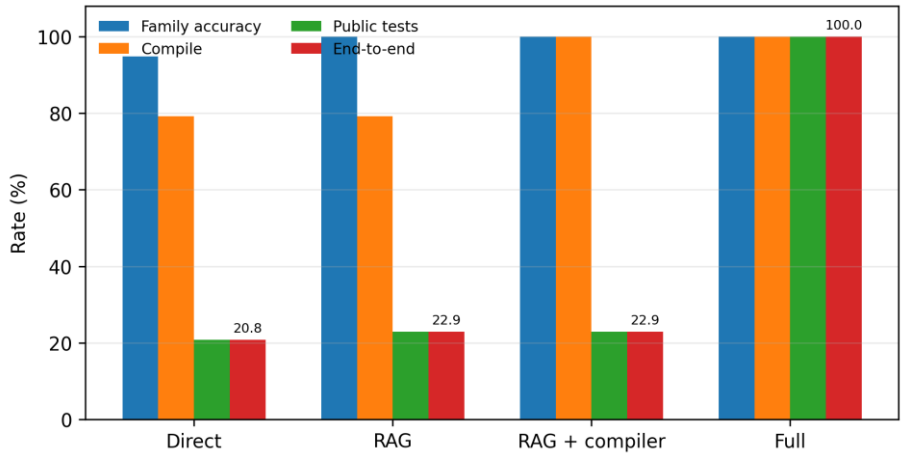

Figure 2. Ablation rates. Values are percentages from actual compiler/execution logs.

The apparent gap in weak baselines where hidden pass exceeds public pass is not leakage: public tests emphasize boundary contracts such as empty collections, equal operands, and zero denominators, whereas hidden cases include randomized ordinary inputs. End-to-end pass is therefore the deployment-relevant metric. The full configuration passes both splits for every task.

6.2. Paired Significance and Robustness

Table 5. Exact Paired Comparisons (E2e)

Comparison	Improved	Regressed	p
Direct → RAG	2	0	0.500
RAG → RAG+Comp.	0	0	1.000*
RAG+Comp. → Full	74	0	1.06×10^{-22}*

*Table 5 uses the end-to-end metric recalculated from the released CSV; when both methods have identical paired outcomes, the exact p-value is 1. The released statistics.json additionally reports held-out-only comparisons: RAG+Compiler→Full improves 63 tasks with no regressions ($p=2.17\times10^{-19}$), and Direct→Full improves 68 with none ($p=6.78\times10^{-21}$). The 95% bootstrap intervals for held-out pass are [19.8, 38.5] for Direct, [21.9, 40.6] for RAG, [25.0, 43.8] for RAG+Compiler, and [100, 100] for Full. The degenerate Full interval reflects a boundary result on 96 controlled tasks, not certainty about a broader population.

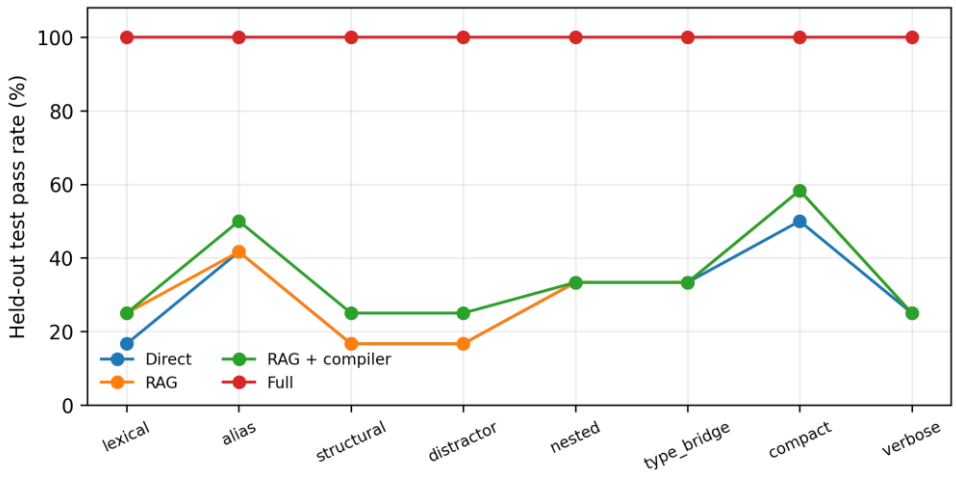

Figure 3. Held-out pass by difficulty. Each point aggregates 12 tasks.

The full system reaches 100% in every difficulty bucket. Weak systems vary from 16.7% to 58.3%, with distractor and structural variants particularly difficult. This pattern supports structural re-ranking and executable selection, but the ceiling also signals insufficient open-world diversity; Section VIII treats it as a primary external-validity threat.

6.3. Illustrative Decision Trace

Consider an Objective-C dictionary lookup returning a fallback. Character and word retrieval nominate dict_default, and the structure analyzer confirms objectForKey/dictionary cues. The preferred deterministic draft may return zero rather than the supplied fallback; it compiles, so compiler feedback cannot distinguish it. The alternative uses dictionary [key] ?? fallback. Public cases containing both present and absent keys reject the preferred draft and select the alternative; six hidden cases then pass. By contrast, for a LegacyInteger draft, type checking fails before execution, and the repair role substitutes Int. The two traces illustrate why compiler evidence and behavioral evidence must remain separate.

Table 6. Example Evidence Flow

Stage	Observed evidence	Decision
Retrieval	objectForKey + fallback	dict_default family
Compiler	both alternatives type-check	no semantic preference
Public tests	wrong draft fails missing-key case	select ?? fallback
Hidden tests	6/6 after selection	evaluation success

6.4. Cost and Diagnostic Behavior

Amortized wall time rises from 35.3 ms for Direct to 39.3 ms for RAG, 56.9 ms for RAG+Compiler, and 190.3 ms for Full. Compiler time dominates: 189.2 of the Full system's 190.3 ms/task. The Full system checks 192 candidates rather than 96 and executes tests for compiling alternatives. This 5.4× latency increase over Direct buys a 79.2-point end-to-end gain in the controlled setting. Batch type checking is therefore essential; launching a compiler process per candidate would measure process overhead rather than framework behavior.

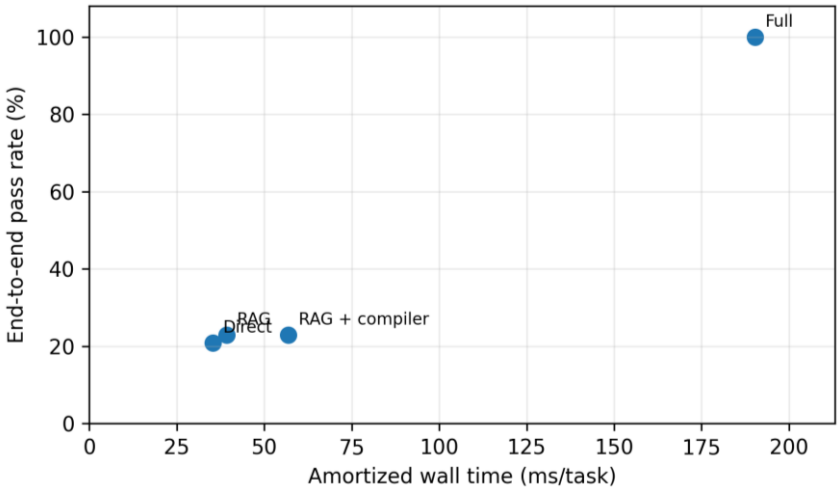

Figure 4. Accuracy–latency trade-off using amortized wall time.

Among RAG+Compiler's selected drafts, 19 LegacyInteger errors and one length member error are repaired; all 20 become compilable. The nominal compile-member and compile-null mutations do not always fire because their trigger substring is absent from some one-line templates; those candidates remain compilable semantic faults. This implementation detail is exposed in the CSV and is a limitation rather than being counted as repair success. More importantly, compilation alone leaves 74 tasks failing at least one public test. The experiment therefore rejects the shortcut of treating compiler acceptance as migration correctness.

7. Discussion

7.1. Why the Components Are Complementary

Retrieval answers "which migration pattern is relevant," the compiler answers "is this Swift admissible," and tests answer "does the candidate satisfy observed behavior." The ablation shows these questions are not interchangeable. Retrieval perfectly classifies the closed schema but cannot prevent invalid Swift. Repair removes diagnosed type/member errors but cannot infer that max was implemented as min. Test arbitration cannot help unless the generator exposes alternatives. The framework's novelty lies less in any single component than in assigning each evidence source a narrow decision right and preserving a trace from input to selected candidate.

7.2. Deployment Path

For a repository deployment, schema memory can be populated from already migrated commits, code review examples, API annotations, and local style rules. The deterministic generator can be replaced by a code model such as CodeT5+ or an instruction-tuned model, while retaining the same contracts. Compilation should run under the project's exact Xcode/SDK configuration; public evidence should include unit tests, snapshot tests, static analyzers, and API-compatibility checks. Candidate budgets can be adapted: low-risk utility methods may use one draft, whereas bridge-heavy classes may justify diverse proposals and human review.

7.3. Artifact Design and Extensibility

The package separates benchmark generation, retrieval/agents, Swift templates, evaluation, and statistics into modules. A learned generator can implement the same render interface, while a project adapter can replace pure-function harnesses with XCTest or Swift Testing. Diagnostic repair rules are data rather than prompt text, making them versionable and reviewable. The design also supports conservative fallbacks: a candidate can be returned as a proposed patch with failed evidence instead of being silently accepted. Rhee et al. offer a complementary and highly relevant perspective through TAPE, where successful and failed trajectories are aligned and curated in an evolving experience pool to improve robustness under distribution shift [34]. This mechanism suggests a promising extension in which validated migrations and failed repair traces continuously refresh ArchMigrate-Swift's schema memory.

7.4. Human Oversight

A passing candidate is not necessarily maintainable, secure, thread-safe, or equivalent under Objective-C runtime reflection. ArchMigrate-Swift is best viewed as a migration assistant that supplies ranked, compiler-checked patches and an evidence report. Human reviewers remain responsible for architecture boundaries, API exposure, concurrency, ownership, performance, and compatibility. The released artifact does not call an LLM and therefore does not evaluate developer trust, prompt security, or model licensing.

8. Threats To Validity

8.1. Construct Validity

The benchmark measures pure-function behavior through finite tests. Real migration includes object identity, inheritance, KVC/KVO, selectors, categories, macros, blocks, error conventions, concurrency, UI lifecycle, and framework availability. Compilation on Linux validates the Swift language and Foundation subset used here, not Xcode build settings or Apple SDK linkage. End-to-end pass is stronger than text similarity but cannot prove equivalence for all inputs.

8.2. Internal Validity

The same fixed generator creates training and test families, so schema leakage is intentional at the family level; only source variants and tests are held out. Candidate defects are deterministic and may favor the public-test arbiter because a correct alternative is included. We expose this mechanism rather than presenting the result as model intelligence. Hidden tests share an executable with public tests to save compile time, but task selection uses public fields only and is implemented before hidden outcomes are consumed. The released source permits inspection of this control.

8.3. External Validity and Statistics

Ninety-six tasks are adequate for paired ablation but small for estimating open-world migration quality. The Full system's 100% causes a boundary confidence interval and should not be extrapolated. The benchmark lacks mined industrial distributions and repository-level dependencies. Future work should build licensed paired corpora from real migrations, stratify by framework and runtime feature, run Xcode/SDK builds, and conduct blinded human review. Statistical p-values quantify paired differences on this benchmark only; they do not measure practical value outside it.

9. Conclusion

ArchMigrate-Swift treats Objective-C-to-Swift migration as an evidence-constrained workflow. Retrieval and structural analysis identify a schema; Swift compilation and diagnostic repair establish language validity; public tests distinguish semantic alternatives; and hidden tests remain evaluation-only. On 96 fixed-seed controlled tasks, the full system achieves 100% compile and end-to-end pass,

while ablations show that retrieval alone, compiler repair alone, and single-candidate generation are insufficient. The key result is not a claim of solved industrial migration, but a reproducible demonstration that compiler feedback and executable arbitration address different failure classes. The accompanying code, generated benchmark, raw outputs, and statistics provide a foundation for replacing deterministic roles with learned agents and evaluating them under real project toolchains.

Reproducibility Statement

The artifact includes all inputs, fixed seeds, source generators, candidate templates, compiler commands, per-task logs, aggregate CSV files, and statistical scripts. Running python run_experiments.py regenerates data and results when Python dependencies and swiftc are available. All numerical claims in this paper are derived from the released result files; no manually entered success count is used as an experimental observation.

Appendix A. Artifact Manifest

The data directory contains train.jsonl, test.jsonl, and manifest.json. The results directory contains per_task_results.csv, summary.csv, derived_summary.csv, statistics.json, run_metadata.json, and environment.json. The root script regenerates the benchmark and runs all four configurations. Raw outputs are retained rather than only the paper tables. The Word paper and figures are generated separately so that changing typography cannot alter experimental values.

Appendix B. Interpretation Checklist

A valid reading of the results must preserve four qualifiers: (1) the benchmark is synthetic and fixed-seed; (2) all test families occur in retrieval memory; (3) the deterministic Full generator includes a correct alternative; and (4) Swift 6.2.1 on Linux does not model Apple SDK integration. Removing any qualifier would overstate the evidence. The intended claim is architectural complementarity under controlled conditions, not autonomous conversion of arbitrary Objective-C applications.

References

[1] Apple Inc. (n.d.). Migrating your Objective-C code to Swift. Apple Developer Documentation. Retrieved July 30, 2026.

[2] Apple Inc. (n.d.). Importing Objective-C into Swift. Apple Developer Documentation. Retrieved July 30, 2026.

[3] Apple Inc. (n.d.). Improving Objective-C API declarations for Swift. Apple Developer Documentation. Retrieved July 30, 2026.

[4] Gregor, D. (2016). SE-0005: Better translation of Objective-C APIs into Swift. Swift Evolution.

[5] Swift.org. (n.d.). API design guidelines. Swift Documentation. Retrieved July 30, 2026.

[6] Rebouças, M., Pinto, G., Ebert, F., Torres, W., Serebrenik, A., & Castor, F. (2016). An empirical study on the usage of the Swift programming language. In Proceedings of the IEEE SANER (pp. 634–638). https://doi.org/10.1109/SANER.2016.66

[7] Domínguez-Álvarez, D., Gorla, A., & Caballero, J. (2022). On the usage of programming languages in the iOS ecosystem. In Proceedings of the IEEE SCAM.

[8] Zak, L. (n.d.). SwiftRewriter: A source-to-source Objective-C to Swift converter. GitHub repository. Retrieved July 30, 2026.

[9] Yandex. (n.d.). Migration toolkit for Swift. GitHub repository. Retrieved July 30, 2026.

[10] Rozière, B., Lachaux, M.-A., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. In Advances in Neural Information Processing Systems 33.

[11] Rozière, B., Zhang, J. M., Charton, F., Harman, M., Synnaeve, G., & Lample, G. (2022). Leveraging automated unit tests for unsupervised code translation. In Proceedings of the ICLR.

[12] Szafraniec, M., Rozière, B., Leather, H. J., Labatut, P., Charton, F., & Synnaeve, G. (2022). Code translation with compiler representations. arXiv preprint arXiv:2207.03578.

[13] Zhu, M., Jain, A., Suresh, K., Ravindran, R., Tipirneni, S., & Reddy, C. K. (2022). XLCoST: A benchmark dataset for cross-lingual code intelligence. arXiv preprint arXiv:2206.08474.

[14] Puri, R., et al. (2021). CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In NeurIPS Datasets and Benchmarks.

[15] Khan, M. A. M., Bari, M. S., Do, X. L., Wang, W., Parvez, M. R., & Joty, S. (2024). XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In Proceedings of the ACL (pp. 6766–6805).

[16] Feng, Z., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of EMNLP (pp. 1536–1547).

[17] Guo, D., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. In Proceedings of the ICLR.

[18] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the EMNLP (pp. 8696–8708).

[19] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified pre-training for program understanding and generation. In Proceedings of the NAACL-HLT (pp. 2655–2668).

[20] Lachaux, M.-A., Rozière, B., Szafraniec, M., & Lample, G. (2021). DOBF: A deobfuscation pre-training objective for programming languages. In Advances in Neural Information Processing Systems 34.

[21] Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems 33 (pp. 9459–9474).

[22] Madaan, A., et al. (2023). Self-Refine: Iterative refinement with self-feedback. In Advances in Neural Information Processing Systems 36.

[23] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems 36.

[24] Wu, Q., et al. (2024). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In Proceedings of the COLM.

[25] Hong, S., et al. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. In Proceedings of the ICLR.

[26] Qian, C., et al. (2024). ChatDev: Communicative agents for software development. In Proceedings of the ACL (pp. 15174–15186).

[27] Huang, D., Bu, Q., Zhang, J. M., Luck, M., & Cui, H. (2023). AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. arXiv preprint arXiv:2312.13010.

[28] Bouzenia, I., Devanbu, P. T., & Pradel, M. (2025). RepairAgent: An autonomous, LLM-based agent for program repair. In Proceedings of the IEEE/ACM ICSE (pp. 2188–2200).

[29] Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[30] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In Advances in Neural Information Processing Systems 35.

[31] Ren, S., et al. (2020). CodeBLEU: A method for automatic evaluation of code synthesis. arXiv preprint arXiv:2009.10297.

[32] Zhao, W., Chen, T., Yang, J. S., & Qiu, L. (2026). AutoML-Pipeline: A RAG-enhanced code generation framework with pre-validation for cloud-native machine learning workflows. IEEE Access, 14, 41932–41945. https://doi.org/10.1109/ACCESS.2026.3673923

[33] Mo, T., Zhang, C., Zou, J., Guo, Z., & Rhee, M. (2026). Self-evolving AI agents with dual memory for automated software testing and bug localization. IEEE Access, 14, 111086–111102. https://doi.org/10.1109/ACCESS.2026.3713401

[34] Rhee, M., Zou, J., Mo, T., Teng, D., & Yang, J.-S. (2026). Enhancing web search agents with self-play contrastive fine-tuning. IEEE Access. https://doi.org/10.1109/ACCESS.2026.3717594