INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH

VOL 04. ISSUE 02 . (2026)

http://ijirjournal.com

Metamorphic-Evidence Arbitration for Compiler-in-the-Loop Objective-C-to-Swift Migration

Chloe Park [1], Kai Davis [2]

[1] Department of Computer Science and Engineering, University of Minnesota Twin Cities, USA

[2] Khoury College of Computer Sciences, Northeastern University, Boston, USA

Corresponding author: *Chloe Park*

Email: *chloepark22@umn.edu*

Abstract.

Cross-language migration requires distinguishing target-language well-formedness from behavior preservation. A compiler can reject ill-typed code but cannot reject a compiling implementation whose semantics diverge on untested inputs. This paper develops an evidence-constrained candidate arbiter that extends the ArchMigrate-Swift compiler-in-the-loop migration workflow with family-specific metamorphic relations. It checks transformations of candidate outputs without relying on reference outputs for the derived inputs. We implemented the system with a deterministic Objective-C-to-Swift migration generator and executed actual Swift 6.2.1 compilation and evaluation. In 64 controlled variants from eight semantic families, public-test arbitration with three sparse public inputs passed 32/64 held-out suites and relation-aware arbitration passed 64/64. Compiler repair raised success from 24/64 to 32/64. Independently compiled Objective-C and Swift reference programs agreed on 1,920/1,920 executions. An adversarial suite of 64 branch-triggered mutants passed the finite metamorphic checks but failed all hidden suites, exposing a decisive coverage limitation. The evidence supports relation-aware selection for specified mutation classes, not general-purpose migration accuracy.

Keywords: *program translation, metamorphic testing, Objective-C, Swift, compiler feedback, candidate selection, reproducible evaluation.*

1. INTRODUCTION

Migration from Objective-C to Swift has remained relevant because the languages differ in nullability, collection behavior, interface import, runtime assumptions, and style. Apple documents incremental migration and imported API adaptation [1], [2]. A syntactically plausible translation can change a boundary condition, default value, or control-flow branch. This problem becomes particularly acute when one-shot translation is replaced by a pool of alternatives: some candidates compile but only a subset preserve the source method's contract. Neural transcompilation has advanced from unsupervised translation [3] to executable-test-aided filtering [4], but a complete engineering workflow needs more than textual similarity. In particular, scarce public tests may fail to expose a mutation hidden outside their sampled inputs. We investigate a narrow but operationally measurable question: can source-derived metamorphic relations improve evidence-based selection without consulting withheld expected outputs?

Our implementation extends ArchMigrate-Swift, the compiler-in-the-loop migration framework of Shang et al. [5]. ArchMigrate-Swift decomposes migration into schema retrieval, structural intent analysis, candidate generation, diagnostic repair, executable criticism and evidence-based arbitration, and it instantiates every role deterministically so that results do not depend on model drift. On its 96-task benchmark, compiler repair raised compilation from 79.2% to 100%, and public-test arbitration over a two-candidate pool raised end-to-end success from 22.9% to 100%. We adopt its separation between candidate generation, target-compiler type checking with diagnostic repair, public-test execution, and final arbitration, together with its deterministic role instantiation. Its public tests are designed around boundary contracts; we study the regime in which public tests are sparse, and modify the arbiter so that a compiling, public-test-passing candidate is also checked against semantic relations derived from the operation family. The contribution is threefold: (i) an executable evidence contract for relation-aware arbitration; (ii) cross-language execution checks under native Clang and Swift compilers, in addition to paired component ablations; and (iii) a pre-specified finite-probe blind-spot experiment that distinguishes successful mutation filtering from universal correctness. The results apply to a transparent, deliberately restricted closed-schema benchmark. Claims about repository-scale migration and LLM coding ability require independent future evaluation.

II. BACKGROUND AND RELATED WORK

A. Cross-Language Translation

TransCoder demonstrated unsupervised mappings between high-level languages [3], and subsequent work used automated tests to improve translation decisions [4]. Compiler representations have been investigated as shared code features [6]. CodeNet [7], XLCoST [8], and xCodeEval [9] are important multilingual resources; their language coverage and evaluation designs differ from Objective-C-to-Swift project migration. CodeBLEU [10] measures structural similarity, but our primary endpoint is actual target execution because multiple implementations can be lexically different yet behaviorally equivalent.

B. Learned Code Representation and Repair

CodeBERT [11] and GraphCodeBERT [12] learn program representations; CodeT5 [13] emphasizes identifiers; PLBART [14] uses sequence-to-sequence pretraining. These works explain how a future learned generator might produce diverse candidates, but representation learning alone does not establish correctness. Self-Refine [15] and Reflexion [16] show the broader usefulness of feedback-driven correction. RepairAgent [17] addresses tool-using program repair. Our controlled artifact instead uses a simple diagnostic rewrite, so the test measures the value of admissibility checks separately from repair-generation intelligence. Execution-guided retrieval and reuse for machine-learning pipeline generation demonstrates the importance of checking generated programs against executable requirements when selecting among retrieved code patterns [27].

C. Agents and Metamorphic Testing

AutoGen [18], MetaGPT [19], ChatDev [20], and AgentCoder [21] motivate specialized role interfaces. The role decomposition used here is implemented by deterministic modules, not conversational agents. Metamorphic testing, introduced by Chen, Cheung, and Yiu [22], addresses oracle limitations by checking relations among executions. Its methodology and limitations have been systematically surveyed by Segura et al. [23]. The new role of metamorphic evidence in our work is not to claim exhaustive validation; it supplies a selection signal unavailable from a small set of exact-output examples.

III. PROBLEM AND EVIDENCE MODEL

Let x denote a source method, K its inferred migration family, $Y=\{y_0,\ldots,y_{m-1}\}$ the candidate Swift methods, C the target compiler, P a set of public example inputs with expected outputs, and H withheld evaluation cases. The selector is not allowed to read H. A public-pass indicator P(y) equals one only when every public output agrees with the independent reference implementation. A compile-pass indicator C(y) equals one only after Swift accepts the selected declaration.

A metamorphic relation R_k consists of pairs or tuples of related inputs and a predicate over the resulting outputs. $R_k(y)=1$ means every specified predicate holds for candidate y. For instance, absolute value satisfies $y(-u)=y(u)$ and $y(u)\geq 0$; a square function satisfies $y(-u)=y(u)$. These checks use the candidate's own outputs and do not query reference outputs for their derived inputs.

Our selection policy first removes invalid or public-failing candidates, then, when metamorphic checking is enabled, removes those violating the family relation. Among remaining candidates the earliest draft wins. A candidate is reported as an abstention if the eligible set is empty. The ranking is intentionally fixed to make counterfactual comparison possible. Crucially, relations are constraints, not proofs: incorrect candidates may satisfy them, and correct candidates may violate an incorrectly specified relation.

A. Selection Rule

Write $E(y)=C(y)\wedge P(y)$, and $M(y)=\wedge_r R_{k,r}(y)$. The original public-test selector chooses $\min\{j:E(y_j)=1\}$; the proposed selector chooses $\min\{j:E(y_j)=1\wedge M(y_j)=1\}$. The selection order gives the faulty first draft the same priority under both conditions, preventing a favorable tie-breaking rule from accounting for any observed gain.

B. Evidence Separation

The compiler stage may inspect Swift text and diagnostics, but not hidden outputs. The public critic checks the three designated public inputs. A distinct relation critic evaluates fixed derived inputs and Boolean equations; its program does not access the reference function. Only the reporting stage computes held-out accuracy against independent Python reference implementations. The artifact logs all selected indices and both splits for inspection.

IV. COMPILER-GUIDED RELATION-AWARE FRAMEWORK

A. Source Schemas and Candidate Creation

The input is a generated Objective-C-style integer method with one of eight behaviors: absolute value, square, parity indicator, clamp to [0,5], doubling, offset by three, cube, and nonnegative floor. Eight variations rename the parameter and method identifier, yielding 64 source functions. Each task has two target templates: an intentionally mutated preferred candidate and a reference-equivalent fallback candidate. This setup evaluates evidence arbitration conditional on a correct alternative already being in the pool; it does not evaluate search recall or an actual LLM. As in ArchMigrate-Swift's controlled setting, the correct alternative is present by construction, and its schema-retrieval stage is replaced by family identities supplied by the generator.

The mutation is selected at the family level and held constant over its eight renamings. Examples include dropping the negative branch, returning $x(x+1)$ instead of x^2, retaining no upper clamp, or replacing $\max(0,x)$ with $|x|$. A correct alternate is known by construction. This makes the benchmark a controlled experiment about distinguishing target programs rather than a broad algorithm discovery benchmark.

B. Compiler Diagnostic Repair

The implementation generates Swift enum-contained methods and invokes swiftc −typecheck on all declarations. An additional pre-repair file injects an unresolved LegacyInteger parameter annotation into both signatures of 16 tasks, producing compiler diagnostics. The repaired file restores Int and passes the same compiler type check. A no-repair configuration discards the 16 affected tasks; with repair, they remain eligible. This reuses the LegacyInteger→Int rule, one of the five diagnostic-grounded edit classes of ArchMigrate-Swift [5], with the actual target compiler, so the compiler stage behaves exactly as in the original framework. Verification-guided multi-agent RTL optimization offers a valuable cross-domain example of directing candidate repair with structured failure evidence and explicit correctness checks [28].

The compiler is run again before the executable is built. Swift and Python versions, elapsed times, source files, and error text are preserved. Repair is intentionally not empowered to modify arithmetic expressions: this preserves the distinction between syntax admissibility and behavioral correctness.

C. Metamorphic Critic

Relations are small, family-specific programs. Absolute value checks symmetry and nonnegativity. Square checks symmetry. Parity checks two-step periodicity and differing parity outputs. Clamp checks output bounds. Doubling checks additivity. Offset checks a unit increment. Cube checks oddness. Nonnegative floor checks idempotence together with equal outputs for a negative input and zero. All checks use fixed integers with no overflow in Int; no external expected-value oracle enters these predicates.

These relations differ substantially in strength. For example, the increment relation accepts x+4 as well as x+3; the public examples must reject the wrong offset. Conversely, a dropped negative branch of absolute value passes nonnegative public inputs but violates symmetry or nonnegativity on relation inputs. Combining the two sources is consequently stronger than using either alone on this task set.

Table 1. OPERATION CONTRACTS USED BY THE RELATION CRITIC

Family	Semantic constraint	Fault targeted
absolute	$f(-u)=f(u)$; $f(u)\geq 0$	negative branch
square	$f(-u)=f(u)$	asymmetric product
parity	$f(u+2)=f(u)$	constant indicator
clamp	$0\leq f(u)\leq 5$	missing cap
double	$f(u+v)=f(u)+f(v)$	quadratic draft
offset	$f(u+1)=f(u)+1$	checked by public
cube	$f(-u)=-f(u)$	even degree
floorzero	$f(f(u))=f(u)$; $f(-3)=f(0)$	absolute draft

Relation inputs are intentionally not the hidden evaluation inputs. The tests are neither inferred automatically from source code nor retrieved from an external benchmark. Rather, each family contract is explicit domain knowledge, encoded as a Boolean predicate that can be independently audited. Distinguishing these assumptions is essential: the performance gain arises because the experimental analyst knows which algebraic relationship should hold, not because an automated agent has discovered program semantics. In future applications, contracts might instead originate from verified specifications, property-based tests such as QuickCheck [24], or reviewed migrations.

We also distinguish relations that characterize a function from those that are merely necessary conditions. Additivity, for instance, is satisfied by infinitely many linear functions; it cannot establish the coefficient two without an anchoring test. Likewise, symmetry alone accepts absolute value, squared value, and many unrelated even functions. The arbiter thus combines relations with known-output public samples. An incorrect candidate is ruled out when either source of evidence detects its failure; success of both remains necessary but not sufficient for universal equivalence.

D. Runtime Architecture and Auditability

A Python coordinator creates Objective-C methods, Swift candidate templates, a pre-repair diagnostic file, a repaired Swift source file, and a main.swift driver. swiftc compiles the driver as one executable; it returns tagged candidate outputs over a fixed range of integer inputs. Python evaluates public predicates, metamorphic checks, and hidden tests in separate passes over this captured output map. Compiler execution and behavioral evaluation thus exercise real Swift methods, not Python reimplementations of those candidates.

Each task/configuration record stores the operation family, selected candidate, public pass, hidden pass, joint end-to-end success, and relation pass. The generator seed is 20261026. By retaining machine-readable logs and a standalone runner, the artifact supports result recomputation, error inspection, and later replacement of the deterministic generator with a learned model.

V. EXPERIMENTAL DESIGN

A. Research Questions

RQ1 asks whether relation-aware selection detects first-draft semantic mutants missed by public examples. RQ2 isolates target-compiler repair. RQ3 measures paired outcomes on disjoint hidden inputs. RQ4 establishes Objective-C/Swift reference agreement through separately compiled source execution, and RQ5 measures how adversarial hidden-trigger branches evade the finite evidence budget.

B. Dataset and Splits

All 64 tasks are synthetic, not mined from published repositories; each of the eight families has eight parameter-name variations and the same family-level candidate mutation. The public inputs are {0,2,4}, deliberately sparser than ArchMigrate-Swift's boundary-focused public tests, so that the experiment probes the regime in which public evidence alone is insufficient. Hidden inputs are {−11,−10,−6,−5,−2,13,14,15,16,17}, which do not coincide with the designated public inputs or with the relation probes in the program. Reference outputs are computed by separately coded Python functions only in the public and evaluation stages. The task JSON retains the Objective-C sources and corresponding expression templates.

Because families, source generation, mutations, and metamorphic constraints are all authored together, this experiment is closed-schema and intentionally favorable to family-specific checks. The fixed split measures a particular failure mode rather than any generalization of training on real-world projects.

C. Configurations

Five configurations are evaluated over the same candidates. Draft-only returns index zero without semantic selection. No-repair/public selects by three public expected-output checks but drops tasks with injected LegacyInteger annotations. Compiler/public is the ArchMigrate-Swift pipeline without retrieval [5]: diagnostic repair followed by public-test arbitration over the same two-candidate pool. Compiler/public/meta additionally imposes relations. Oracle upper bound selects using the withheld evaluation outputs and is shown only as a ceiling; it is not a deployable system and plays no role in the proposed selector.

D. Metrics and Statistical Procedure

For each configuration we record the fraction passing all public cases, all ten hidden cases, and both splits. We also count relation satisfaction. The primary paired comparison is end-to-end success of compiler/public against compiler/public/meta over identical task IDs. We report discordant pairs and a task-level exact two-sided McNemar/binomial p-value [25]. Because tasks share eight family-level mutations, the 64 records are not independent experimental units. A second exact calculation aggregates by the eight families (four gains, zero losses; two-sided p=0.125). Neither statistic is interpreted as an effect estimate for independently sampled repositories.

E. Execution Environment

Experiments were run with Swift 6.2.1, Clang 17.0.0 in Objective-C mode, and Python on x86_64 Linux. The pre-repair type-check failed with unresolved LegacyInteger, the repaired type-check completed successfully, and a compiler-produced executable generated the outputs from which all metrics were calculated. One run recorded a repaired type-check time of 172.8 ms, a complete build time of 871.8 ms, and an executable runtime of 7.5 ms. These are run-level batch measurements and are not presented as a realistic interactive LLM-service latency.

VI. RESULTS

A. Main Ablation

Table II reports observations computed from results.csv. The draft-only configuration never passes all ten hidden inputs because each family's preferred draft is semantically altered. Public tests reject some mutations but cannot distinguish others. The no-repair/public setting passes 24 of 64 tasks (37.5%); enabling the compiler repair adds eight successes (32/64, 50.0%). Adding metamorphic filtering increases end-to-end

success to 64/64 (100%) on the same fixed candidates. With its boundary-focused public tests, ArchMigrate-Swift's public arbitration alone reached 100% on its own benchmark [5]; under the three sparse public inputs used here the same arbiter reaches 50.0%, and the metamorphic critic recovers the remaining 32 tasks. The relation critic therefore complements the original arbiter, supplying the behavioral evidence that sparse public tests cannot. The oracle ceiling also reaches 64/64, because every task includes a correct fallback.

Table 2OBSERVED RESULTS ON 64 CONTROLLED TASKS

Method	Public	Hidden	End-to-end
draft only	32/64	0/64	0/64
unrepaired public	48/64	24/64	24/64
comp. public (ArchMigrate-Swift)	64/64	32/64	32/64
comp. public meta	64/64	64/64	64/64
oracle upper bound	64/64	64/64	64/64

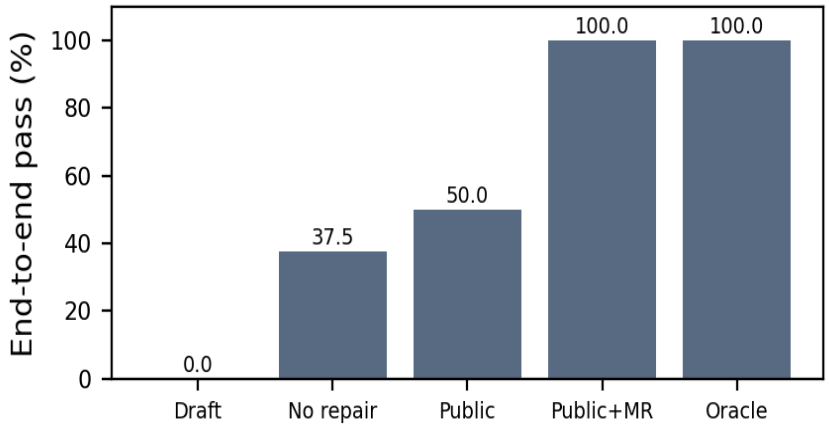


Figure 1. End-to-end success on 64 controlled synthetic tasks (from execution logs).

B. Paired Outcome Analysis

Relative to compiler/public, relation-aware arbitration changes 32 failed tasks into successes without any regression, giving an exact two-sided p-value of 4.66×10^{-10} under a task-level paired binomial model. The observed difference is +50 percentage points. At the family level the gain is concentrated in four mutated operations rather than 32 independent defect types. Consequently the evidence supports the diagnostic value of the selected metamorphic predicates in this controlled setting, not an established average gain for unknown project populations.

C. Error Mechanism and Trace

Consider a source method computing absolute value. The preferred Swift candidate returns x unchanged on either side of the conditional. Because public inputs are nonnegative, public evaluation accepts this candidate. The metamorphic critic evaluates a negative/positive pair and finds both a symmetry failure and a nonnegativity failure, removing the first candidate. Its fallback uses abs(x), compiles, passes all public inputs, and also passes the ten hidden inputs. This is a genuine candidate selection intervention; the critic does not synthesize the correct expression or consult hidden expected outputs.

For the offset family, the first candidate returns x+4 instead of x+3. That candidate satisfies the unit-increment metamorphic relation but fails the public exact-output tests; public evidence is sufficient. For the clamp family, a candidate without an upper bound can survive nonnegative in-range public inputs but violates the

relation output bound at larger values. These examples show why semantic relation predicates are complementary to exact example tests, rather than substitutes for them.

D. Compiler Contribution

The injected LegacyInteger annotation is rejected by swiftc. Once the unresolved annotation is replaced with Int, the repaired source type-checks and the executable compiles. The 16 syntax-affected task entries are excluded by the no-repair configuration; eight of their candidates would otherwise pass hidden evaluation under public arbitration. Rehabilitating these entries accounts for the improvement from 24/64 to 32/64. The study does not measure general-purpose compiler repair or additional diagnostic classes.

E. Runtime and Reproduction

Compilation dominates the short run: one observed batch build required approximately 871.8 ms, compared with 7.5 ms for executing all generated candidate/input pairs. All variants share this precomputed executable output table, so these times cannot be allocated causally to an individual arbitration configuration. The evidence critic itself consists of arithmetic comparisons over already collected outputs. The released runner can regenerate source, compilation checks, candidate outputs, task-level CSV, and aggregate statistics without a model endpoint.

F. Independently Compiled Source-Language Cross-Check

A separate source_check executable compiled all 64 generated source functions using clang -x objective-c. For every function, Clang and the Swift reference candidate were executed on all 30 integers from −11 through 18. Their outputs agreed in 1,920/1,920 paired executions, with zero observed differences. The raw comparison is source_crosscheck.csv; the compiler invocation and concrete sources are retained in robustness_experiment.py and objective_c_sources.m. This verifies the restricted arithmetic migration semantics over the tested range; because no messages or Foundation classes occur, it is not a test of Objective-C object-runtime interoperability.

G. Adversarial Finite-Probe Negative Control

To interrogate the apparent perfect score, we created 64 separate Swift mutants, each replacing its reference expression f(x) with x=t ? f(x)+1 : f(x), where t is one of {13,14,15,16,17}. These trigger points were deliberately selected outside all public and relation probes but inside the withheld evaluation domain. Candidate index zero is the mutant; index one remains correct. Both candidates were compiled and executed using Swift, and selection still uses only public and relation checks. All 64 mutants pass public and metamorphic predicates, so both selectors retain the incorrect first draft; all 64 then fail their hidden suites. The experiment is a deliberately constructed negative control rather than a randomized estimate of defect frequency.

Table 3ACTUAL EXECUTION AND COVERAGE STRESS TESTS

Validation	Observed	Interpretation
Clang vs Swift reference	1,920 / 1,920	agreement on 30 inputs x 64 tasks
Ordinary public+MR	64 / 64	known family mutations filtered
Adversarial public+MR	0 / 64	held-out trigger branches survive
Adversarial MR pass	64 / 64	finite relations miss hidden branches

This contrast is central to interpreting the method. Relations can identify faults that violate their algebraic predicates at the chosen probes; they cannot establish semantic equivalence when the difference lies outside those probes. The stress suite is intentionally worst-case, but it refutes any interpretation of 64/64 on the ordinary suite as a reliability guarantee. Cross-compiled agreement of the reference target does not alter this conclusion: arbitration quality and source-to-reference semantics are separate quantities.

VII. DISCUSSION

A. Why Relation-Based Evidence Matters

A compiler validates that an expression is permitted by the target language. Small public tests validate pointwise behavior at selected inputs. A metamorphic predicate validates a relationship over multiple executions without knowing each expected result. The evidence channels are orthogonal in their failure coverage: invalid Swift can satisfy a mathematical relation in abstract, and a syntactically valid incorrect algorithm can pass three examples. A robust arbiter should therefore accumulate distinct forms of evidence rather than repeatedly ask a generator for another answer. Our relation checker applies constraints only after public-test filtering. This ordering avoids promoting a function that satisfies a weak algebraic relation while getting the basic operation wrong. A real deployment could associate relation provenance with a migration schema or verified legacy test suite, rank relation strength, and abstain when no viable candidate remains. Relation trust becomes a first-class engineering concern: a mistaken contract can eliminate correct software.

B. Scope and External Validity

The experiment is intentionally small and synthetic. Renaming a parameter eight times does not provide eight independent semantic challenges. The restricted source code is compiled in Objective-C mode and executed on Linux, but it uses a C-like arithmetic subset without Objective-C object features; the Swift targets are checked independently. The benchmark omits Foundation bridging, nullable objects, dynamic dispatch, KVC/KVO, blocks, memory ownership, and Apple SDK dependencies. It also assumes the correct candidate appears in a two-item pool, a strong condition that may not hold for real LLM decoding.

Further, relations were manually defined for the same eight operation families used to synthesize candidate defects. The ordinary-suite 100% success characterizes only those authored mutation classes, as the adversarial zero-success control concretely demonstrates. Evaluation on independently collected developer migrations, fault taxonomies and candidate pools from several actual code models is required before claims about migration reliability or transfer to other language pairs can be made.

C. Threats to Construct and Internal Validity

Finite input tests cannot prove functional equivalence, even for integer programs. Metamorphic constraints are incomplete and occasionally satisfied by multiple distinct functions. Candidate scoring uses family identities supplied by the generator, so retrieval accuracy is not under test. The repair ablation uses a single synthetically injected compilation error class. Public test inputs and hidden inputs are disjoint, and hidden expected outputs are evaluated only after selection, but relations use deliberately chosen probe points rather than a blind external test design.

Single-machine elapsed time varies with process startup and system load; we retain timings as evidence of executed processes without making fine-grained performance comparisons. The exact paired p-value treats task records as units although shared mutation families induce clustering. A future family-level replication, randomized fault sampling, and appropriately clustered uncertainty intervals (rather than naive task-level bootstrap resampling [26]) would provide stronger estimates.

D. Sensitivity to Benchmark Construction

The task variants deliberately preserve semantics while changing identifiers and method names; therefore the 64 records should not be read as 64 independently sampled software requirements. If the eight families are used as the conceptual units, the experiment exhibits four family-level gains and zero family-level regressions (exact two-sided $p=0.125$). This clustering changes the interpretation of formal significance: the task-level McNemar statistic is useful for reproducing the exact paired comparison, whereas broader inferential claims require multiple independently sampled programs per semantic family. The observed success rate is also sensitive to mutation choice. A mutant that preserves the tested relation but differs elsewhere will not be filtered unless another public or relational check exercises its divergence.

Candidate diversity is another prerequisite. When a pool contains only an incorrect variant, relation checking can abstain but cannot create a correct solution. In this artifact the reference-equivalent fallback is explicitly included for every task to make selection efficacy identifiable. This is analogous to separating proposal recall from reranker accuracy in information retrieval: improving the reranker does not ensure that the candidate generator covers the desired answer. A future factorial design should vary pool size, candidate recall, constraint quality, and public-test coverage independently.

E. Deployment Implications

The method is suitable as a validation layer downstream of any producer that emits multiple candidate migrations. A learned model can replace template generation if its outputs are still passed through an exact target compiler, public-test harness, and relation predicates from trusted domain knowledge. Integrated into the full ArchMigrate-Swift pipeline, including its schema-retrieval stage, this separation would allow future studies to vary language model, retrieval memory, and candidate budget while holding the evidence critic fixed. A realistic migration service should return evidence traces and abstentions to a human reviewer rather than silently equate a passing finite suite with proof of equivalence.

VIII. CONCLUSION

We tested a compiler-guided, relation-aware candidate arbitration method for controlled Objective-C-to-Swift migration. Its distinguishing contribution is to add family-specific metamorphic predicates as a decision constraint after compilation and public example tests. In the ordinary 64-task synthetic suite, relation-aware arbitration increased hidden-suite successes from 32 to 64, and compiler repair improved the public-only baseline from 24 to 32. A separately compiled Clang-versus-Swift execution audit showed 1,920/1,920 agreements. However, all 64 deliberately constructed hidden-trigger candidates evaded the existing public and metamorphic probes, demonstrating a sharp limitation of finite evidence. The experiment establishes a reproducible mechanism for using complementary validity and behavioral evidence, and provides an inspectable starting point for future open-world, LLM-driven migration evaluation.

REPRODUCIBILITY

Run python experiment.py, then python robustness_experiment.py, in the artifact directory with Swift 6.2.1, Clang 17, Python 3 and SciPy. The script regenerates tasks.json, candidate Swift files, compiler_diagnostics.txt, results.csv, summary.csv, statistics.json, source_crosscheck.csv, stress_results.csv, and robustness_statistics.json. Task-wise selected candidate indices are retained; no external LLM, network service, or proprietary dataset is required.

REFERENCES

Apple Inc., “Migrating your Objective-C code to Swift,” Apple Developer Documentation, accessed Oct. 2026.

Apple Inc., “Importing Objective-C into Swift,” Apple Developer Documentation, accessed Oct. 2026.

B. Rozière, M.-A. Lachaux, L. Chanussot, and G. Lample, “Unsupervised translation of programming languages,” in Adv. Neural Inf. Process. Syst., vol. 33, 2020.

B. Rozière et al., “Leveraging automated unit tests for unsupervised code translation,” in Proc. ICLR, 2022.

W. Shang, J.-S. Yang, G. Ding, and S. Yang, “ArchMigrate-Swift: A compiler-in-the-loop retrieval-augmented multi-agent framework for Objective-C-to-Swift migration,” Trans. Comput. Sci. Intell. Syst. Res., vol. 13, pp. 275–285, 2026, doi: 10.62051/khg1am73.

F. Zhou, “Moment-constrained side-force allocation for wind-assisted merchant ships: Hydrodynamic performance of retractable anti-leeway fins and flap rudders,” Innovation and Technology Studies, vol. 3, no. 1, pp. 27–41, 2026.

R. Puri et al., “CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks,” in NeurIPS Datasets Benchmarks, 2021.

M. Zhu et al., “XLCoST: A benchmark dataset for cross-lingual code intelligence,” arXiv:2206.08474, 2022.

M. A. M. Khan et al., “xCodeEval: An execution-based large-scale multilingual multitask benchmark for code understanding, generation, translation and retrieval,” in Proc. ACL, pp. 6766–6805, 2024, doi: 10.18653/v1/2024.acl-long.367.

S. Ren et al., “CodeBLEU: A method for automatic evaluation of code synthesis,” arXiv:2009.10297, 2020.

Z. Feng et al., “CodeBERT: A pre-trained model for programming and natural languages,” in Findings EMNLP, pp. 1536–1547, 2020.

J. Wang, Y. Ge, J. Liu, and B. Jiang, “Early warning of liquidity stress in U.S. banks with seasonally aligned frailty hazards and delayed-feedback conformal alarms,” Journal of Trends in Finance and Economics, vol. 1, no. 1, pp. 85–100, 2024.

Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in Proc. EMNLP, pp. 8696–8708, 2021, doi: 10.18653/v1/2021.emnlp-main.685.

W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Unified pre-training for program understanding and generation,” in Proc. NAACL-HLT, pp. 2655–2668, 2021, doi: 10.18653/v1/2021.naacl-main.211.

A. Madaan et al., “Self-Refine: Iterative refinement with self-feedback,” in Adv. Neural Inf. Process. Syst., vol. 36, 2023.

N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, “Reflexion: Language agents with verbal reinforcement learning,” in Adv. Neural Inf. Process. Syst., vol. 36, 2023.

I. Bouzenia, P. Devanbu, and M. Pradel, “RepairAgent: An autonomous, LLM-based agent for program repair,” in Proc. IEEE/ACM ICSE, 2025, doi: 10.1109/ICSE55347.2025.00157.

Q. Wu et al., “AutoGen: Enabling next-gen LLM applications via multi-agent conversation,” in Proc. COLM, 2024.

S. Hong et al., “MetaGPT: Meta programming for a multi-agent collaborative framework,” in Proc. ICLR, 2024.

C. Qian et al., “ChatDev: Communicative agents for software development,” in Proc. ACL, pp. 15174–15186, 2024, doi: 10.18653/v1/2024.acl-long.810.

X. Yue and R. Zhu, “An explainable hybrid machine learning framework for cash-flow-conditioned multi-horizon liquidity risk early warning: An SME-oriented benchmark study,” World Journal of Management Science, vol. 2, no. 1, pp. 63–72, 2024.

T. Y. Chen, S. C. Cheung, and S. M. Yiu, “Metamorphic testing: A new approach for generating next test cases,” Tech. Rep. HKUST-CS98-01, 1998.

S. Segura, G. Fraser, A. B. Sánchez, and A. Ruiz-Cortés, “A survey on metamorphic testing,” IEEE Trans. Softw. Eng., vol. 42, no. 9, pp. 805–824, 2016, doi: 10.1109/TSE.2016.2532875.

J. Chen, X. Han, X. Zhang, and Y. Yang, “Exact concept-level explanations for early detection of financial misstatements,” Eurasia Journal of Science and Technology, vol. 5, no. 2, pp. 34–48, 2023.

Q. McNemar, “Note on the sampling error of the difference between correlated proportions or percentages,” Psychometrika, vol. 12, no. 2, pp. 153–157, 1947, doi: 10.1007/BF02295996.

B. Efron and R. J. Tibshirani, An Introduction to the Bootstrap. Chapman & Hall, 1993.

W. Zhao, T. Chen, B. Wang, and Z. Shen, “PIPE-RAG: An Execution-Guided Retrieval and Reuse Framework for Reliable Machine Learning Pipeline Code Generation,” WEP Transactions on Computer Science and Intelligent Systems Research, vol. 13, pp. 286–297, 2026.

H. Zhang, “A Verification-Guided Multi-Agent Framework for Reliable RTL Generation and Optimization,” WEP Transactions on Computer Science and Intelligent Systems Research, vol. 13, pp. 260–274, 2026.