SemaMig-Bench: Evaluating Behavioral Preservation in COBOL-to-Java Migration

George Weale
Strust
george.weale@columbia.edu

Abstract

Autonomous coding agents can translate repositories, but most benchmarks do not ask whether a generated target preserves the behavior of a running legacy system. SemaMig-Bench is a source-oracle benchmark for COBOL-to-Java migration. Each task supplies an original COBOL repository, a Java execution contract, public examples, and an empty target workspace. After an agent produces a native Java candidate, held-out workloads run independently against pinned COBOL and target-only Java environments. Contract-scoped observations are normalized and compared; a task passes only when every required case agrees. Verifier admission requires alternative correct implementations, anti-wrapper controls, domain-specific semantic mutants, deterministic execution, and independent review. Evaluation reports task-macro BP@1 and BP@k, build-to-behavior gaps, first-divergence categories, cost, and uncertainty across controlled-harness and native-product tracks. The benchmark measures behavioral preservation rather than source resemblance or compilation alone.

1 Introduction

A translated repository can compile and still be wrong. In COBOL-to-Java migration, decimal scale, fixed-width records, file-status branches, state updates, and batch ordering are externally visible behavior. Compilation and target-side unit tests can therefore reward a plausible implementation that changes the system being migrated.

SemaMig-Bench asks one question: given a COBOL repository, a Java execution contract, public examples, and a fixed resource budget, can an agent build a native Java implementation that matches the executable source on unseen workloads? During construction, the agent may inspect and probe the source. During grading, the same hidden inputs are executed independently against a pinned COBOL environment and a target-only Java environment. A contract-scoped comparator grades externally visible behavior, and a task passes only when every required case matches.

RepoMod-Bench is the closest existing benchmark: it evaluates repository modernization using hidden, implementation-agnostic interface tests [3]. SemaMig-Bench adds a narrower legacy-migration construct. The source is executed on the same held-out workload as the candidate, observations are tailored to COBOL migration hazards, and verifier strength is checked with alternative correct implementations and semantic mutants. The design also borrows the original-task and functional-verifier discipline of DeepSWE [2] while targeting reconstruction rather than repository change.

SemaMig-Bench organizes original, practitioner-reviewed systems across diagnostic, workflow, and system tiers. It separates public development tasks from hosted evaluation tasks and grades both through the same source-oracle contract. The contribution is threefold:

- A COBOL-to-Java corpus model stratified by migration hazard and implementation horizon.
- A paired source-oracle evaluator with target-only candidate execution and exact task-level scoring.
- A verifier-admission and repeated-rollout protocol for comparing models and coding products.

2 SemaMig-Bench

2.1 Task formulation

An agent receives four visible artifacts: a COBOL source repository, a declared Java build and execution contract, a behavior contract, and public examples. It writes a native Java implementation into an empty target workspace under a fixed tool and compute budget. The behavior contract names the observable surfaces that matter, such as exit status, output records, database deltas, ordered side effects, messages, and timeout state. It does not prescribe Java classes, frameworks, or internal architecture.

The agent may inspect and execute the source while constructing its candidate when the evaluation track permits source probing. It never receives hidden workloads, reference Java code, or private comparator fixtures. At grading time, the Java artifact is rebuilt and executed without the COBOL source or runtime.

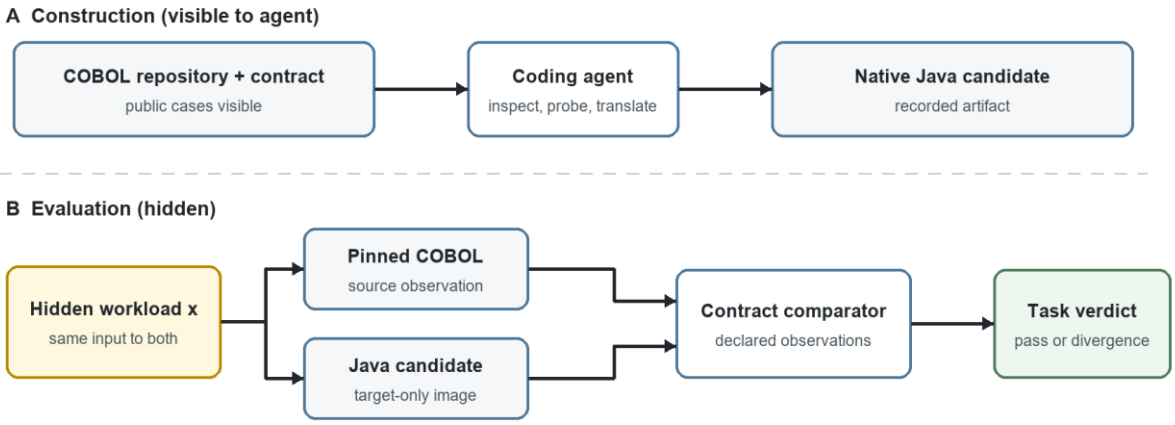

Figure 1. Construction uses visible source materials. Grading runs the same hidden workload against isolated COBOL and Java environments, then compares only contract-declared observations.

2.2 Corpus design

Table 1. Corpus tiers by implementation horizon.

Tier	Typical source size	Evaluation unit
Diagnostic	200-1,000 LOC	One or two programs isolating a semantic hazard
Workflow	1,000-5,000 LOC	Multi-file business workflow with shared records or state
System	5,000-20,000 LOC	Multiple programs, copybooks, and batch or service boundaries

The tiers describe implementation horizon, not difficulty. A short packed-decimal task may be harder than a regular multi-file workflow. Results are therefore reported overall, by tier, and by primary semantic hazard.

Table 2. Primary semantic strata used for construction and error analysis.

Stratum	Source behaviors	Representative migration defect
Decimal arithmetic	PICTURE scale, COMP-3, sign, overflow, rounding	Binary floating point or incorrect rounding mode
Text and encoding	Fixed width, spaces, EBCDIC, collation	Trimming, wrong code page, changed sort order
Record layout	REDEFINES, OCCURS, overlays, copybooks	Incorrect offsets, bounds, or shared storage
Control flow	PERFORM THRU, GO TO, 88 levels, size errors	Branch inversion or skipped fall-through behavior
Batch I/O	Files, status codes, dates, step order, restart	Wrong order, boundary date, or swallowed error state
State and composition	CALL context, shared data, messages, idempotency	Missing, duplicated, or reordered state change

Tasks are newly authored rather than copied from customer estates. Each must be plausible to a legacy practitioner, contain coherent business rules, and expose behavior through declared interfaces. Original authorship makes licensing tractable and reduces known target-solution leakage, but it does not make synthetic systems equivalent to production estates. Provenance, generation assistance, licenses, similarity checks, and reviewer history are recorded for every task.

The benchmark separates a public development set from a hosted evaluation set. Development tasks include complete tests and reference traces. Active evaluation workloads remain hosted to limit test leakage; retired tasks, runners, container definitions, comparator code, and analysis scripts are released so that past verdicts can be independently recomputed from raw observations.

2.3 Task admission

A task enters the frozen suite only when it passes four gates:

1. **Determinism.** Repeated source executions under the same image, seed, locale, and fixture produce identical contract observations.
2. **Acceptance breadth.** A reviewed reference implementation and at least one structurally different correct Java implementation pass all cases.
3. **Verifier strength.** Null candidates, public-case hard coding, source wrappers, and reviewed semantic mutants fail.
4. **Independent review.** The instruction, behavior contract, hidden partitions, normalizers, and failure evidence are approved by reviewers who did not author the candidate.

A fixed-width payroll workflow serves as the end-to-end calibration task. It covers packed-decimal signs, rounding, maximum field widths, invalid dates, rejection ordering, and restart behavior, exercising the complete task schema and evaluator before the same gates are applied to wider workflows.

3 Verification and Scoring

3.1 Paired source execution

For a hidden input `x`, the evaluator runs source and target independently:

```
os = O(Run(Ps, x; Es));  ot = O(Run(Pt, x; Et))
```

`Ps` and `Pt` are the source and target programs, `Es` and `Et` are pinned environments, and `O` extracts only contract-declared observations. The comparator then applies versioned normalizers and evaluates `C(Ns(os), Nt(ot))`. Normalization is narrow: a task may decode COMP-3 values through a declared schema or transcode a named field, but it may not discard fixed-width spaces or replace source-defined decimal rules with a tolerance chosen after seeing failures.

Each case returns MATCH, DIVERGE, or INCONCLUSIVE. MATCH requires agreement on every required observation. DIVERGE means comparable states were reached and at least one required observation differs. INCONCLUSIVE is reserved for evaluator-side failures or invalid source preconditions. Candidate compilation errors, crashes, timeouts, prohibited dependencies, and resource overruns are task failures rather than inconclusive outcomes.

3.2 Metrics

A rollout passes a task only when the candidate builds and every required hidden case matches. The primary metric, Behavioral Pass@1 (BP@1), is the macro-average task success rate across independent rollouts, with every task weighted equally. BP@k reports the fraction of tasks solved at least once in exactly `k` attempts. The benchmark does not claim either name as a new statistical estimator; the contribution is the source-oracle task verdict beneath it.

Diagnostic reporting includes build success, hidden-case agreement, the gap between build success and BP@1, results by tier and semantic hazard, first-divergence category, wall time, tokens, agent steps, and cost. Confidence intervals use a task-cluster bootstrap that keeps all rollouts for a sampled task together. Pairwise claims use paired task outcomes and effect sizes with intervals; overlapping rows are not described as definitively ordered.

3.3 Integrity and verifier quality

The graded candidate runs in a target-only image with no COBOL executable, source files, undeclared system binary, or network access. Dependencies are allowlisted and recorded. Artifact inspection and adversarial cases reject subprocess wrappers, foreign-function calls into the source, vendored ports, remote delegation, and lookup tables over public examples.

Verifier adequacy is measured with domain-specific mutants representing plausible migration defects: changed rounding, lost signs, implicit trimming, incorrect OCCURS bounds, branch inversion, swallowed file status, reordered output, missing state writes, and replayed batch effects. Equivalent mutants are removed through review. A task is not admitted while a declared high-severity mutant survives. Each results release reports mutants

attempted, mutants removed, mutation kill rate, alternate-correct-implementation acceptance, and blinded reviewer disagreement on sampled passing and failing trials.

4 Experimental Protocol

4.1 Tracks and baselines

The controlled-model track holds the coding harness fixed: system prompt, shell and file interface, context policy, workspace, dependency policy, and wall-clock limit. It measures model capability under a shared scaffold. The native-product track evaluates coding products with their default tools and context management. Its results are reported separately because product scaffolding is part of the system under test.

Table 3. Controlled baselines and the feedback available to each configuration.

ID	Configuration	Feedback available	Purpose
B0	Direct translation	Source and contract only	One-shot lower bound
B1	Compiler-guided	Build and compiler errors	Syntax and build repair
B2	Public-case iteration	Build plus public cases	Visible-test iteration
B3	Inspect-only agent	B2 plus source reading	No source execution
B4	Source-probing agent	B3 plus source execution	Primary probing ablation

The key ablation is source inspection versus active source probing. Compiler-only and public-example baselines separate syntax repair from behavioral discovery, while a shared agent harness supports controlled model comparisons. A file-level COBOL translation model may be added on diagnostic tasks as a calibration baseline but is not compared directly with repository agents on workflow and system tasks.

4.2 Frozen evaluation

Each evaluation release declares task counts, strata, admission gates, model identifiers, budgets, rollout count, metrics, intervals, exclusions, and subgroup analyses before scoring begins. Evaluations use multiple independent rollouts per task and configuration. Hidden cases and severity labels are frozen before model results are inspected; corrections require a new benchmark version and reruns of affected rows.

The principal analyses ask four questions: how often candidates compile but fail behavioral scoring; whether active source probing improves preservation; which semantic hazards produce first divergences; and whether automated verdicts agree with blinded expert review. Cost and reliability are reported alongside task success. Every results release includes the frozen corpus profile, baseline runs, verifier mutation statistics, and failure analysis.

5 Related Work

SWE-bench established repository-level issue resolution with executable tests [7], and DeepSWE strengthens that recipe with original tasks, shared scaffolding, hand-written functional verifiers, complete trajectories, and independent grading audits [2]. LiveCodeBench addresses contamination through time-windowed tasks [8], while SpecBench shows that agents can satisfy visible tests through memorization shortcuts [9]. These works motivate original authorship, held-out evaluation, and verifier auditing, but they do not evaluate legacy-system reconstruction.

RepoTransBench evaluates complete repository translation across language pairs using executable target tests [4]. COBOL-JavaTrans evaluates bidirectional COBOL and Java translation in a HumanEval-derived setting [5]. RepoMod-Bench is closest to SemaMig-Bench: it uses hidden, implementation-agnostic tests over standardized interfaces and documents wrapper shortcuts [3]. SemaMig-Bench does not claim to be the first modernization benchmark. Its core contribution is paired execution against a live COBOL source, COBOL-specific observation contracts, semantic strata, mutation-audited verifiers, and target-only anti-wrapper isolation. AgentModernize likewise treats modernization as behavioral preservation, but evaluates a multi-agent workflow on eight scenarios rather than defining a shared model benchmark [6].

The source-oracle design follows differential testing, which exposes disagreements by executing multiple implementations on the same input [10]. Because automated oracles remain a central testing bottleneck [11], the source is treated as a practical pseudo-oracle for preservation rather than proof of correctness. Mutation testing

provides negative controls for verifier strength [12]. STRUST-v1 applies related contract-driven differential verification to ten controlled COBOL-to-Java pairs, but evaluates the verifier rather than migration agents [1].

6 Limitations, Governance, and Conflict of Interest

Preservation is not intent. A source system may contain defects or obsolete behavior. SemaMig-Bench measures preservation under a declared contract; intentional corrections require a separate change-approval task.

Finite workloads are not equivalence proofs. Passing establishes agreement only over the declared observations and held-out workload. Generated partitions, curated boundary cases, and mutation testing improve resolution without establishing universal semantic equivalence.

Portable COBOL is not a mainframe. GnuCOBOL supports reproducible public tasks but does not reproduce every IBM Enterprise COBOL option, EBCDIC environment, JCL step, VSAM mode, CICS interaction, or DB2 behavior. Each task must state its dialect and subsystem coverage, and results must not be extrapolated to unsupported estates.

Synthetic tasks underrepresent organizational complexity. Practitioner review can improve realism but cannot reproduce undocumented conventions, production dependencies, or decades of change history. Hosted evaluation also trades full offline reproducibility for resistance to evaluation-time test leakage.

The transparency model combines public task schemas, runners, comparators, container recipes, development tasks, retired evaluation tasks, and analysis code with hosted active workloads. Signed result bundles expose task versions, artifact digests, aggregate counts, exclusions, timing, and cost. Strust sponsors the benchmark and works on behavioral evaluation; this conflict is disclosed. Primary verdicts must be independently recomputable from raw source and candidate observations through an open reference comparator, and external stewards approve task admission and contested verdicts.

7 Conclusion

SemaMig-Bench specifies a focused test of agentic legacy migration: can a native Java candidate preserve the contract-relevant behavior of an executable COBOL source on unseen workloads? Its credibility rests on paired source execution, COBOL-specific observation contracts, mutation-audited verifiers, target-only grading, and transparent repeated evaluation. By fixing the evidence contract and publishing the evaluation record, the benchmark makes behavior preservation reviewable across models and coding products.

References

[1] G. Weale, "STRUST-v1: Slice-Based Differential Verification for Large COBOL-to-Java Migrations," Strust, 2026. https://strust.us/strust-v1.pdf

[2] W. Huang, C. Lee, L. Tng, and S. Ge, "DeepSWE: Measuring Frontier Coding Agents on Original, Long-Horizon Engineering Tasks," arXiv:2607.07946, 2026. https://arxiv.org/abs/2607.07946

[3] X. Li, N. Ben-Israel, Y. Raz, B. Ahmed, D. Serebro, and A. Raux, "RepoMod-Bench: A Benchmark for Code Repository Modernization via Implementation-Agnostic Testing," arXiv:2602.22518, 2026. https://arxiv.org/abs/2602.22518

[4] Y. Wang et al., "RepoTransBench: A Real-World Benchmark for Repository-Level Code Translation," arXiv:2412.17744, 2024. https://arxiv.org/abs/2412.17744

[5] A. T. V. Dau, S. H. Tan, J. Yang, N. D. Q. Bui, and A. T. Nguyen, "COBOL-Coder: Domain-Adapted Large Language Models for COBOL Code Generation and Translation," arXiv:2604.03986, 2026. https://arxiv.org/abs/2604.03986

[6] S. N. Ahmed and M. Galib, "AgentModernize: Preserving Business Logic in Legacy Modernization with Multi-Agent LLMs and Behavioral Specification Graphs," arXiv:2605.17535, 2026. https://arxiv.org/abs/2605.17535

[7] C. E. Jimenez et al., "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" International Conference on Learning Representations, 2024. https://openreview.net/forum?id=VTF8yNQM66

[8] N. Jain et al., "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code," International Conference on Learning Representations, 2025. https://arxiv.org/abs/2403.07974

[9] B. Zhao, D. Srikanth, Y. Wu, and Z. Jiang, "SpecBench: Measuring Reward Hacking in Long-Horizon Coding Agents," arXiv:2605.21384, 2026. https://arxiv.org/abs/2605.21384

[10] W. M. McKeeman, "Differential Testing for Software," Digital Technical Journal, vol. 10, no. 1, pp. 100-107, 1998.

[11] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," IEEE Transactions on Software Engineering, vol. 41, no. 5, pp. 507-525, 2015. doi:10.1109/TSE.2014.2372785

[12] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, 2011. doi:10.1109/TSE.2010.62