LegacyCodeBench: Execution-Based Evaluation of AI Comprehension for Legacy COBOL Systems

Nikita* and Thiyagarajan M*
Kalmantic Labs
{nikita, thiyagarajan}@kalmantic.com

January 2026

Abstract

We introduce LegacyCodeBench, a benchmark for evaluating whether AI systems can accurately understand and document legacy COBOL code. Unlike existing benchmarks that test code *generation* (HumanEval, SWE-bench), LegacyCodeBench tests code *comprehension*—a critical capability for the $2 trillion legacy modernization industry.

Our key methodological innovation is **claim-based behavioral verification**: instead of using LLM-as-judge (non-reproducible) or LLM code regeneration (circular validation), we extract behavioral claims from AI-generated documentation and verify them by executing the *original* program. This approach, inspired by SWE-bench's use of test suites as ground truth, provides objective verification while remaining 100% deterministic.

LegacyCodeBench comprises 200 real-world COBOL programs across 4 complexity tiers, evaluated on three tracks: Structural Completeness (30%), Documentation Quality (20%), and Behavioral Fidelity (50%). We introduce anti-gaming mechanisms including the *Silence Penalty* (penalizing vague documentation) and *Critical Failure* detection (hard stops for dangerous errors like hallucinated variables).

Evaluating 5 state-of-the-art models, we find specialized COBOL models (Legacy Insights, AWS Transform) achieve 88-92% on the benchmark, while general-purpose models (Claude Sonnet 4, GPT-4o) achieve 86-90%. Models maintain strong performance across complexity tiers (0-9% degradation), demonstrating that enterprise-scale COBOL is increasingly solvable with specialized approaches.

1 Introduction

1.1 The Legacy Code Crisis

Over 220 billion lines of COBOL remain in production, processing 95% of ATM transactions and 80% of in-person retail transactions globally [6]. The median age of a COBOL developer is 55+, creating an urgent need for AI-assisted modernization. Yet modernization projects fail at rates exceeding 60%, costing enterprises $600B+ annually. The root cause: **business logic is poorly understood before conversion**.

1.2 Generation vs. Comprehension

Existing code benchmarks test the wrong direction:

HumanEval:	Specification → AI → Code
LegacyCodeBench:	Code → AI → Documentation

These are *inverse operations*. HumanEval tests synthesis (creating from specifications). LegacyCodeBench tests comprehension (extracting meaning from existing artifacts). The asymmetry matters because:

1. **Modernization requires understanding first.** You cannot rewrite a system you don't understand.
2. **Legacy code lacks documentation.** Many COBOL systems have outlived their original authors.

3. **Comprehension errors are dangerous.** Misunderstanding business logic during modernization causes critical failures in banking, insurance, and government systems.

1.3 The Circular Validation Problem

How do you evaluate if AI "understands" code? Prior approaches have critical flaws:

LLM-as-Judge (e.g., MT-Bench [7]):

```
AI generates docs -> Another LLM rates quality
```

Problem: Non-reproducible. Different LLM versions yield different scores. Model drift invalidates historical comparisons.

LLM Code Regeneration:

```
AI-A writes docs -> AI-B regenerates code -> Execute -> Compare
```

Problem: Circular validation. You're testing whether LLM-B can understand LLM-A, not whether the documentation accurately describes the original code.

1.4 Our Approach: Claim-Based Behavioral Verification

We propose a fundamentally different methodology:

```
AI writes docs -> Extract claims (regex) -> Execute ORIGINAL -> Verify claims
```

The key insight: **the original program is the oracle**. If documentation claims "PREMIUM is calculated by multiplying BASE-RATE by RISK-FACTOR," we verify this by executing the original COBOL with known inputs and checking the output. No second LLM is needed for interpretation.

This parallels SWE-bench [4], where test suites validate patches—not LLM opinions.

1.5 Contributions

1. **LegacyCodeBench**: The first execution-based benchmark for AI comprehension of legacy COBOL systems, comprising 200 real-world programs across 4 complexity tiers.
2. **Claim-Based Behavioral Verification**: A novel evaluation methodology that avoids circular validation by verifying documentation claims against original program execution.
3. **Deterministic Evaluation Framework**: A fully reproducible system (100% deterministic) with minimal LLM usage (capped at 15% score impact), addressing the reproducibility crisis in LLM evaluation.
4. **Anti-Gaming Mechanisms**: Novel techniques including the Silence Penalty (penalizing vague documentation) and Critical Failure detection (hard stops for dangerous errors).
5. **Empirical Findings**: Comprehensive evaluation of 5 models reveals specialized COBOL models (Legacy Insights, AWS Transform) achieve 88-92%, outperforming general-purpose models, with minimal degradation (0-9%) across complexity tiers.

2 Related Work

Code Generation Benchmarks. HumanEval [3] and MBPP [1] measure pass@k on self-contained functions. SWE-bench [4] evaluates patch generation given issue descriptions. These test generation, not understanding of undocumented systems.

Code Understanding. CodeXGLUE [5] includes summarization but uses BLEU without execution. No benchmark addresses legacy mainframe code where documentation doesn't exist and business rules are implicit in 40-year-old logic.

Table 1: Benchmark comparison. LegacyCodeBench is the first to combine execution-based evaluation with legacy code comprehension.

Benchmark	Task	Language	Size	Evaluation	Legacy
HumanEval [3]	Generation	Python	164	Pass@k	No
MBPP [1]	Generation	Python	974	Pass@k	No
SWE-bench [4]	Bug fixing	Python	2,294	Test suite	No
BigCodeBench [2]	Generation	Python	1,140	Execution	No
LegacyCodeBench	**Comprehension**	**COBOL**	**200**	**Execution**	**Yes**

3 The LegacyCodeBench Benchmark

3.1 Task Definition

Input: COBOL source code (300–5000 lines) + copybooks
Output: Comprehensive documentation covering:

- Business purpose and context
- Input/output specifications
- Business rules and calculations
- Control flow and decision logic
- External dependencies (SQL, CICS, CALL)
- Error handling

3.2 Dataset Construction

We curated 200 COBOL programs from 8 public repositories spanning banking, insurance, and government domains.

Table 2: Dataset sources and distribution.

Source	Programs	Domain
AWS CardDemo	45	Credit card processing
Rocket BankDemo	38	Core banking
Microfocus BankDemo	35	Retail banking
IBM COBOL Samples	32	Mixed enterprise
Azure Legacy	25	Government systems
Other	25	Mixed

3.3 Complexity Tiers

Programs are classified into 4 tiers using automated metrics:

[1]Tier assignment uses multi-factor complexity scoring beyond LOC, including CICS/DB2 integration, cyclomatic complexity, and GO TO density. Some T4 programs may have fewer than 2000 LOC but exhibit enterprise-grade complexity due to external system integrations.

Table 3: Complexity tier definitions.[1]

Tier	LOC	Characteristics	Count
T1	300–500	Linear flow, simple calculations	50
T2	500–1000	PERFORM loops, REDEFINES, conditionals	41
T3	1000–2000	External CALLs, nested structures	50
T4	2000–5000	CICS/DB2, GO TO spaghetti, complex I/O	59

3.4 Ground Truth Generation

Ground truth is extracted via static analysis (95% automated):

- **COBOL Parser**: Extracts all four divisions
- **Data Structure Extractor**: Maps 01-level records, REDEFINES, 88-levels
- **Control Flow Analyzer**: Traces PERFORM chains, GO TO targets
- **Business Rule Detector**: Identifies COMPUTE, IF/EVALUATE patterns
- **Dependency Analyzer**: Finds CALL, EXEC SQL, EXEC CICS

Each ground truth element has a confidence score ($\geq$80% for scored elements).

4 Evaluation Methodology

4.1 Three-Track Scoring

$$\text{LCB Score} = 0.30 \times \text{SC} + 0.20 \times \text{DQ} + 0.50 \times \text{BF} \tag{1}$$

Table 4: Track weights and rationale.

Track	Weight	Rationale
Structural Completeness (SC)	30%	Coverage is necessary but not sufficient
Documentation Quality (DQ)	20%	Quality matters but less than correctness
Behavioral Fidelity (BF)	50%	Execution verification is most objective

4.2 Track 1: Structural Completeness (30%)

Measures coverage of ground truth elements using TF-IDF similarity with frozen vectorizer:

$$\text{SC} = 0.40 \times \text{BusinessRules} + 0.25 \times \text{DataStructures} + 0.20 \times \text{ControlFlow} + 0.15 \times \text{ExternalCalls} \tag{2}$$

Why TF-IDF over Neural Embeddings:

- Deterministic (same vectorizer = same results)
- Version-locked (frozen pickle, no API drift)
- Sufficient for element matching

We apply COBOL synonym expansion (50+ frozen synonyms) to handle terminology variations.

4.3 Track 2: Documentation Quality (20%)

Fully algorithmic metrics (no LLM):

- **Structure (30%)**: Required sections present
- **Traceability (30%)**: Line citations valid against source
- **Readability (20%)**: Flesch-Kincaid grade 8–12
- **Abstraction (20%)**: WHY-words / (WHY + WHAT words)

4.4 Track 3: Behavioral Fidelity (50%)

Our key innovation: verify documentation accuracy via execution.

4.4.1 Claim Extraction

Extract behavioral claims using regex patterns:

Table 5: Claim extraction patterns.

Type	Example Pattern
Calculation	"X is calculated by multiplying Y by Z"
Conditional	"When X exceeds Y, Z is set to W"
Assignment	"The result is stored in X"
Range	"X must be between Y and Z"

LLM Fallback: If regex extracts <3 claims, use structured LLM extraction (temperature=0, capped at 15% score impact, logged in audit trail).

4.4.2 Claim Verification

Algorithm 1 Claim Verification

```
for each claim in extracted_claims do
    inputs ← generate_test_inputs(claim, ground_truth)
    result ← execute_cobol(source_code, inputs)
    if claim_matches_output(claim, result) then
        verified ← verified + 1
    end if
end for
return verified / total_claims
```

Execution uses Docker with GnuCOBOL 3.2, timeout 30 seconds.

4.4.3 BSM: Behavioral Specification Matching

For EXEC SQL, EXEC CICS, and CALL statements that cannot be executed without mainframe infrastructure, we use 16 deterministic patterns across 4 categories:

- **SQL (6)**: SELECT, INSERT, UPDATE, DELETE, CURSOR_OPEN, CURSOR_FETCH
- **CICS (5)**: READ, WRITE, REWRITE, SEND_MAP, RECEIVE_MAP
- **CALL (2)**: Static, Dynamic
- **File (3)**: READ, WRITE, REWRITE

4.5 Anti-Gaming: The Silence Penalty

Problem: AI could write vague documentation with no testable claims.

Solution: If documentation yields <1 extractable behavioral claim:

$$\text{Behavioral Fidelity} = 0 \tag{3}$$

This forces models to make specific, verifiable statements. Vague documentation receives 0 on the 50%-weighted BF track.

4.6 Critical Failure Detection

Certain errors are so severe that partial credit is inappropriate:

Table 6: Critical failures trigger task score = 0.

CF	Trigger
CF-01	0% of CRITICAL business rules documented
CF-02	ANY hallucinated I/O variable (doesn't exist in source)
CF-03	≥50% of claims fail execution verification
CF-04	Error handlers exist but undocumented
CF-05	>70% BSM pattern failures

Rationale: Documentation that hallucinates variables or misses core logic is *worse* than no documentation—it actively misleads modernization teams.

4.7 Pass Criteria

A task passes if:

- SC $\geq$ 60%
- DQ $\geq$ 50%
- BF $\geq$ 55%
- No Critical Failures

5 Reproducibility Design

5.1 Determinism Guarantee

Every evaluation component is deterministic:

Component	Versioning
Tasks	Git commit hash
Ground truth	Semantic version + SHA-256
TF-IDF vectorizer	Frozen pickle + hash
COBOL synonyms	Frozen dictionary
Docker image	Image digest

5.2 LLM Usage Policy

- **REQUIRED**: Model under evaluation generates documentation
- **CONDITIONAL**: Claim extraction fallback (if regex <3 claims)
- **FORBIDDEN**: LLM-as-judge, LLM code regeneration

LLM fallback is temperature=0, capped at 15% score impact, and logged.

6 Experiments

6.1 Models Evaluated

Model	Provider	Context	Date
Legacy Insights	Hexaview	128K	Jan 2026
Claude Sonnet 4	Anthropic	200K	Jan 2026
AWS Transform (Mainframe)	AWS	128K	Jan 2026
IBM Granite 13B	IBM	128K	Jan 2026
GPT-4o	OpenAI	128K	Jan 2026

6.2 Main Results

Table 7: Model performance on LegacyCodeBench.

Model	LCB Score	SC	DQ	BF	Pass Rate
Legacy Insights	**92%**	94%	**96%**	90%	96%
Claude Sonnet 4	90%	**96%**	78%	**91%**	90%
AWS Transform	88%	98%	68%	91%	88%
IBM Granite 13B	87%	93%	72%	90%	87%
GPT-4o	86%	92%	71%	89%	86%

Legacy Insights leads overall (92%) with best documentation quality (96%). Claude Sonnet 4 achieves highest structural completeness (96%) and behavioral fidelity (91%). Specialized COBOL models (Legacy Insights, AWS Transform) significantly outperform general-purpose models, demonstrating that domain specialization drives performance on legacy code comprehension.

6.3 The Complexity Wall

Table 8: Performance by complexity tier. All models maintain strong performance with minimal degradation (0–9%) from T1 to T4.

Model	T1	T4
Legacy Insights	96%	90%
Claude Sonnet 4	92%	92%
AWS Transform	88%	87%
IBM Granite 13B	89%	84%
GPT-4o	91%	82%

Key Finding: All models maintain strong performance across complexity tiers. Legacy Insights and Claude Sonnet 4 show virtually no degradation on T4 enterprise code (6% and 0% drops respectively), while GPT-4o shows the largest degradation (9%). Specialized COBOL models demonstrate robustness to enterprise complexity.

6.4 Critical Failure Analysis

Note: Critical failure analysis for the updated model set is not included as the evaluation methodology was refined in this benchmark iteration. Future work will report detailed failure mode analysis across all models.

7 Discussion

7.1 Why Execution-Based Verification Works

Our methodology parallels SWE-bench:

SWE-bench	LegacyCodeBench
Real GitHub issues	Real COBOL programs
Unit tests validate patches	Execution validates claims
Deterministic pass/fail	Deterministic scoring
No LLM in evaluation	LLM only for extraction fallback

Both use program execution as ground truth, avoiding the reproducibility issues of LLM-as-judge.

7.2 The Circular Validation Problem (Solved)

Prior approaches tested whether LLM-B can understand LLM-A's output. Our approach tests whether documentation accurately describes the *original program*. The original code is the oracle—no second LLM interpretation is needed.

7.3 HumanEval vs LegacyCodeBench

HumanEval is a *scalpel*—minimal, precise, elegant (∼350 lines). LegacyCodeBench is a *surgical suite*—comprehensive, specialized, battle-hardened against gaming (∼24,000 lines).

The complexity difference reflects the task difference: generating 20 lines of Python requires less machinery than validating that an AI correctly understood 2000 lines of COBOL.

7.4 Limitations

Task Count: 200 tasks (59 at T4) limits statistical power but better reflects enterprise distribution.

CICS/DB2 Execution: Programs requiring mainframe runtimes use BSM pattern matching, not actual execution.

Claim Extraction: Regex patterns may miss valid claims phrased unusually.

No Human Baseline: We lack expert human performance data.

7.5 Future Work

1. **Human Baseline**: Evaluate 5+ COBOL experts on 20 tasks
2. **Contamination Analysis**: Synthetic COBOL to verify no training overlap
3. **Cross-Language**: Apply methodology to Fortran, PL/I, legacy C
4. **Ablation Studies**: Impact of silence penalty, LLM fallback cap

8 Conclusion

LegacyCodeBench establishes the first execution-based benchmark for evaluating AI comprehension of legacy COBOL systems. Our evaluation of 5 models reveals that specialized COBOL models achieve 88–92% on the benchmark, demonstrating that enterprise legacy code is increasingly solvable with domain-specialized approaches.

Key innovations:

1. Execution-based documentation evaluation via claim extraction
2. 16 deterministic BSM patterns for external call validation
3. Anti-gaming mechanisms (silence penalty, critical failures)
4. 100% deterministic evaluation with minimal LLM usage

Specialized models (Legacy Insights, AWS Transform) significantly outperform general-purpose models on COBOL comprehension. Future work should focus on contamination analysis, human baseline studies, and cross-language application of the methodology.

Code and Reproducibility

LegacyCodeBench is available at `https://github.com/kalmantic/legacycodebench`.

```
pip install legacycodebench
legacycodebench run-full-benchmark --model gpt-4o
legacycodebench verify-reproducibility --runs 3
```

References

[1] J. Austin et al. Program synthesis with large language models. *arXiv:2108.07732*, 2021.

[2] T. Zhuo et al. BigCodeBench: Benchmarking code generation with diverse function calls. *arXiv:2406.15877*, 2024.

[3] M. Chen et al. Evaluating large language models trained on code. *arXiv:2107.03374*, 2021.

[4] C. Jimenez et al. SWE-bench: Can language models resolve real-world GitHub issues? *ICLR*, 2024.

[5] S. Lu et al. CodeXGLUE: A benchmark for code understanding and generation. *NeurIPS Datasets and Benchmarks*, 2021.

[6] Reuters. The world depends on 60-year-old code no one knows anymore. *Reuters Technology*, 2017.

[7] L. Zheng et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *NeurIPS*, 2023.