ESOLANG-BENCH: EVALUATING GENUINE REASONING IN LARGE LANGUAGE MODELS VIA ESOTERIC PROGRAMMING LANGUAGES

Anonymous authors
Paper under double-blind review

ABSTRACT

We present EsoLang-Bench, a benchmark revealing fundamental limitations in how large language models (LLMs) leverage in-context learning (ICL). Frontier models achieving 85–95% accuracy on standard code benchmarks (HumanEval, MBPP) score only 0–11% on esoteric programming languages with scarce training data. **Notably, few-shot prompting yields no statistically significant improvement over zero-shot** ($p = 0.505$), contradicting assumptions about ICL enabling adaptation to novel domains. Our analysis indicates that ICL primarily activates training priors rather than enabling genuine learning. Despite this limitation, **self-scaffolding with direct interpreter feedback outperforms multi-agent approaches**, and **agentic systems achieve 2–3× improvement** through interpreter feedback loops with efficient context management. These findings have implications for understanding LLM generalization to out-of-distribution domains.

1 INTRODUCTION

Large language models have achieved strong performance on code generation benchmarks, with frontier models such as GPT-5.2 and Gemini-3 Pro attaining 85–95% accuracy on HumanEval and MBPP (Chen et al., 2021; Austin et al., 2021). However, a fundamental question remains: *do these results reflect genuine programming understanding, or primarily pattern matching against training distributions?*

We investigate this question through esoteric programming languages, which are Turing-complete but have minimal web presence. Unlike mainstream languages, esoterics such as Brainfuck, Befunge-98, Whitespace, Unlambda, and Shakespeare provide a natural test for generalization: their specifications can be provided in-context, yet training contamination is minimal.

The in-context learning hypothesis (Brown et al., 2020) posits that providing specifications and examples should enable model adaptation to new tasks. Under this hypothesis, few-shot prompting with working examples should yield significant improvement over zero-shot baselines. This work empirically tests this assumption and presents findings on the conditions under which different prompting approaches succeed or fail.

2 WHY THIS BENCHMARK?

2.1 THE BENCHMARK GAMING PROBLEM

Current code generation benchmarks increasingly suffer from data contamination (Zhang et al., 2024; Sainz et al., 2023) and design flaws that allow models to achieve high accuracy through pattern matching rather than genuine reasoning (Gupta et al., 2024). This phenomenon reflects Goodhart's Law (Goodhart, 1984): "when a measure becomes a target, it ceases to be a good measure."

EsoLang-Bench addresses this limitation by targeting domains where optimization is **economically irrational**. Including esoteric language data in pre-training would: (1) provide no deployment value, (2) incur high collection costs, and (3) potentially degrade performance on mainstream tasks. This

Table 1: Comparison of static vs. OOD benchmark paradigms.

ASPECT	STATIC	**OOD (OURS)**
CONTAMINATION RISK	HIGH	MINIMAL
GAMING INCENTIVE	HIGH	NONE
EVALUATES	RETRIEVAL	REASONING
HUMAN ALIGNMENT	WEAK	STRONG

economic disincentive for gaming ensures the benchmark measures genuine capability rather than optimization artifacts.

2.2 MEASURING TRANSFERABLE REASONING

This benchmark evaluates **general transferable reasoning**: the ability to apply computational primitives (loops, conditionals, state management) to unfamiliar syntactic domains. By requiring models to solve problems in languages with scarce training data, EsoLang-Bench isolates genuine algorithmic understanding from pattern retrieval. This evaluation paradigm **parallels human programming language acquisition**, where developers learn new languages through documentation, interpreter feedback, and iterative experimentation rather than memorized examples.

To construct such a benchmark, we select five esoteric languages based on criteria ideal for evaluating genuine reasoning: **(1) Turing Completeness**, all can express any computable function; **(2) Paradigm Diversity** spanning memory-tape (Brainfuck), 2D execution (Befunge-98), invisible syntax (Whitespace), combinatory logic (Unlambda), and natural-language syntax (Shakespeare); **(3) Interpreter Availability** enabling test-time learning for feedback loops via self-scaffolding; and **(4) Data Scarcity** with 1,000–100,000$\times$ less training data than Python.

3 EXPERIMENTAL SETUP

Dataset. 80 problems across four difficulty tiers (Easy, Medium, Hard, Extra-Hard), each with 6 test cases. Problems span arithmetic, string manipulation, number theory, and algorithmic tasks (see Appendix C for full categorization).

Models. Five frontier models: GPT-5.2, O4-mini-high, Gemini-3 Pro, Qwen3-235B, Kimi-K2.

Prompting Strategies. We evaluate five approaches:

- **Zero-Shot**: Language documentation + problem specification
- **Few-Shot**: Above + 3 solved examples
- **Self-Scaffolding (S-S)**: Iterative refinement with direct interpreter feedback (1 API call/iter, max 5 attempts)
- **Textual Self-Scaffolding (TSS)**: Coder generates code, critic analyzes failures in natural language (2 API calls/iter, max 5 attempts)
- **ReAct (Re)**: Planner creates algorithm, coder translates, critic evaluates and provides textual feedback (3 API calls/iter, max 5 attempts) (Yao et al., 2023)

Agentic Systems. We additionally evaluate GPT-5.2 Codex and Claude Code (Opus 4.5) with direct interpreter access.

4 RESULTS

4.1 ICL PROVIDES NO IMPROVEMENT

Table 2 presents our principal finding: **few-shot prompting yields no statistically significant improvement over zero-shot** (Wilcoxon signed-rank test, $p = 0.505$). Despite receiving three complete working examples demonstrating language syntax and idioms, models do not leverage this information effectively.

Table 2: Zero-shot and few-shot accuracy (%) across models and languages. Few-shot provides no systematic improvement.

	BF		**Bef**		**WS**		**Unl**		**Shk**	
Model	0-S	F-S	0-S	F-S	0-S	F-S	0-S	F-S	0-S	F-S
GPT-5.2	2.5	2.5	2.5	**8.8**	0	0	0	0	**2.5**	1.2
O4-mini	2.5	**3.8**	**6.2**	7.5	0	0	0	0	1.2	1.2
Gemini	2.5	3.8	5.0	3.8	0	0	0	0	1.2	1.2
Qwen	2.5	1.2	0	0	0	0	0	**1.2**	0	0
Kimi	0	0	2.5	1.2	0	0	0	0	1.2	1.2

Table 3: Scaffolding strategy accuracy (%) by model and language. S-S = Self-Scaffolding (1 API call); TSS = Textual Self-Scaffolding (2 API calls); Re = ReAct (3 API calls). Best per row in **bold**.

	Brainfuck			**Befunge**			**Whitespace**			**Unlambda**			**Shakespeare**		
Model	S-S	TSS	Re	S-S	TSS	Re	S-S	TSS	Re	S-S	TSS	Re	S-S	TSS	Re
GPT-5.2	**6.2**	3.8	5.0	**11.2**	10.0	8.8	0	0	0	**1.2**	0	0	**2.5**	2.5	1.2
O4-mini	**5.0**	2.5	3.8	**10.0**	6.2	7.5	0	0	0	0	0	0	**1.2**	1.2	0
Gemini	**5.0**	3.8	3.8	**7.5**	6.2	7.5	0	0	0	0	0	0	**1.2**	0	0
Qwen	**2.5**	1.2	2.5	0	0	0	0	0	0	**1.2**	0	0	**1.2**	0	0
Kimi	0	0	0	0	0	0	0	0	0	0	0	0	**1.2**	0	0

This result contradicts the hypothesis that ICL enables adaptation to novel domains, and instead supports the interpretation that ICL success on standard benchmarks primarily reflects activation of training priors rather than genuine in-context learning (Min et al., 2022).

4.2 Complete Failure Beyond Easy Tier

Beyond the Easy tier, all models achieve **0% accuracy on Medium, Hard, and Extra-Hard problems** regardless of prompting strategy, including agentic systems with interpreter access. These same problems are trivially solvable in Python (95%+ accuracy), demonstrating a fundamental lack of transferable reasoning: models cannot apply known algorithmic primitives to unfamiliar syntactic domains.

4.3 Scaffolding Strategy Comparison

Table 3 demonstrates that **self-scaffolding with direct interpreter feedback consistently outperforms multi-agent approaches**. GPT-5.2 achieves 11.2% on Befunge-98 with self-scaffolding ($p < 0.01$ vs. zero-shot). Interpreter feedback loops provide unambiguous ground-truth signals enabling incremental debugging. In contrast, TSS and ReAct introduce intermediary LLM interpretation that adds noise when all components lack domain knowledge: the critic cannot provide useful feedback about esoteric syntax it has never seen, compounding errors rather than correcting them.

4.4 Agentic System Results

Table 4 demonstrates that agentic systems with direct interpreter access achieve **2–3× improvement** over non-agentic approaches, with Codex attaining 13.8% accuracy on Brainfuck. This improvement stems from efficient context management: (1) context persistence through structured logging and selective fetching of relevant prior attempts, and (2) dynamic retrieval of question-specific solved examples rather than static demonstrations, mitigating the "lost-in-the-middle" phenomenon (Liu et al., 2024).

5 Error Analysis

Figure 1 reveals a diagnostic pattern:

Table 4: Agentic system accuracy (%) with direct interpreter access.

System	Brainfuck	Befunge-98	Average
Codex (Agentic)	**13.8%**	**8.8%**	**11.2%**
Claude Code (Opus 4.5)	12.5%	8.8%	10.6%

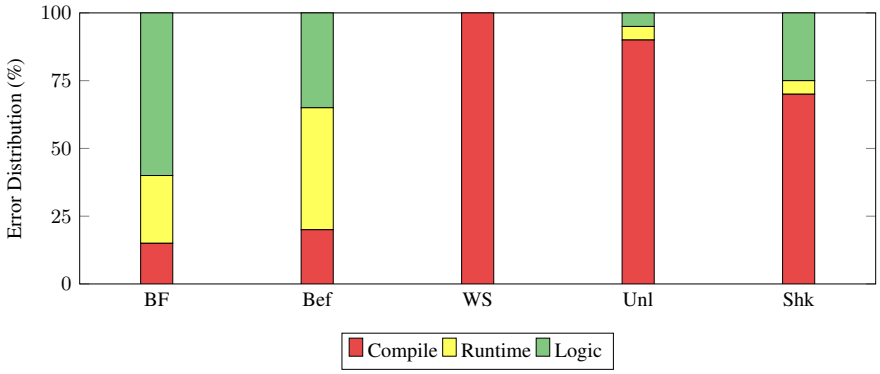

Figure 1: Error distribution by language (GPT-5.2 zero-shot). Whitespace and Unlambda show near-total compile failure; Brainfuck shows primarily logic errors.

- **Higher-resource languages** (Brainfuck, Befunge-98): With 500–2,000 GitHub repositories, models acquire syntax (15–20% compile errors) but fail on semantics (35–60% logic errors)
- **Ultra-low-resource languages** (Whitespace, Unlambda, Shakespeare): With <200 repositories, models exhibit high compile failure (70–100%), struggling to generate valid syntax

This binary pattern provides a clear diagnostic for the boundary between partial and absent pre-training coverage.

6 Key Takeaways

For practitioners building systems for OOD domains:

(1) Skip few-shot curation. ICL cannot activate priors that do not exist. For OOD tasks, zero-shot with documentation matches few-shot while saving prompt engineering effort.

(2) Prefer direct feedback over multi-agent reasoning. When domain knowledge is absent, each additional LLM layer amplifies errors. Ground-truth signals provide cleaner gradients than LLM-generated critiques.

(3) Invest in context management. Structured logging with selective retrieval of prior attempts and dynamic question-specific examples outperforms static few-shot and full history approaches.

7 Conclusion

EsoLang-Bench establishes a contamination-resistant benchmark for evaluating **transferable reasoning**: the capacity to apply learned computational primitives to unfamiliar syntactic domains. By utilizing Turing-complete esoteric languages where benchmark optimization is economically irrational, this work provides an evaluation paradigm that parallels human learning through documentation, interpreter feedback, and iterative experimentation. Our findings reveal fundamental limitations in frontier models: despite strong performance on mainstream benchmarks, these models fail to transfer reasoning skills across syntactic boundaries, suggesting that current architectures may rely more heavily on pattern matching than genuine algorithmic understanding.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

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

Charles AE Goodhart. Problems of monetary management: The UK experience. *Monetary Theory and Practice*, pp. 91–121, 1984.

Vipul Gupta, David Pantoja, Candace Ross, Adina Williams, and Megan Ung. Changing answer order can decrease MMLU accuracy. *arXiv preprint arXiv:2406.19470*, 2024.

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

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11048–11064, 2022.

Oscar Sainz, Jon Campos, Iker García-Ferrero, Julen Etxaniz, Oier Lopez de Lacalle, and Eneko Agirre. NLP evaluation in trouble: On the need to measure LLM data contamination for each benchmark. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 10776–10787, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2023.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Charlotte Zhuang, Dylan Slack, Qin Lyu, Sean Hendryx, Russell Kaplan, Michele Lunati, and Summer Yue. A careful examination of large language model performance on grade school arithmetic. *arXiv preprint arXiv:2405.00332*, 2024.

Reproducibility Statement

All code, prompts, interpreters, and evaluation scripts will be released upon publication. Problems utilize deterministic test cases with exact string matching. Interpreter implementations for all five esoteric languages are provided.

Ethics Statement

This work evaluates publicly available models on programming tasks. Benchmark results should not be extrapolated to claims about general AI capabilities. The objective is calibrated assessment by identifying capability boundaries on out-of-distribution tasks. AI assistants were used for research and literature survey during the preparation of this paper.

A Benchmark Overview Figure

Figure 2 provides a visual overview of the EsoLang-Bench evaluation framework.

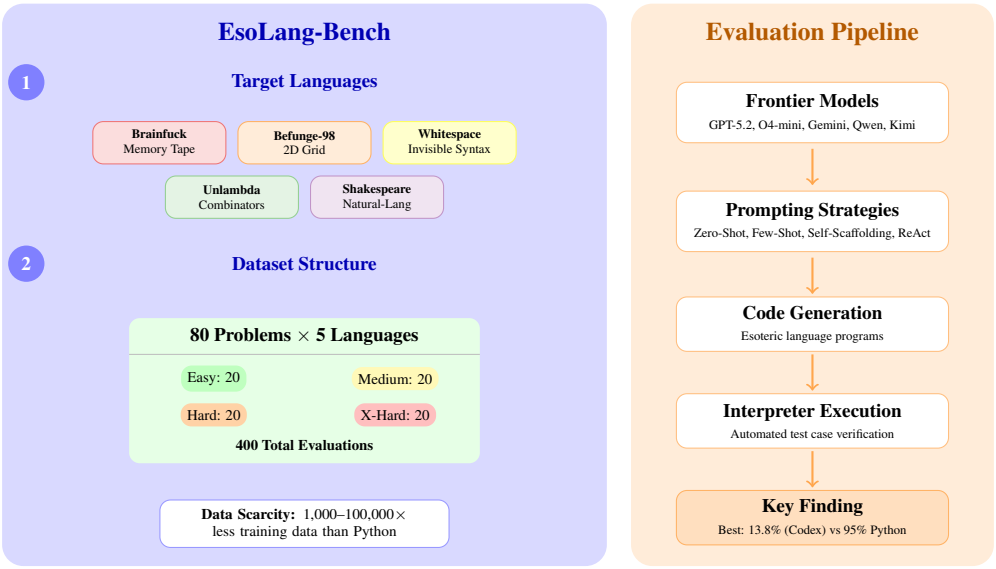

Figure 2: **EsoLang-Bench Overview.** *Left:* The benchmark comprises five esoteric programming languages spanning diverse computational paradigms, with 80 problems across four difficulty tiers (400 total evaluations). *Right:* Evaluation pipeline testing five frontier models across multiple prompting strategies, with automated interpreter-based verification.

B Esoteric Language Details

Brainfuck (1993): Created by Urban Müller as a challenge to build the smallest possible compiler. The language has only 8 commands (`>`, `<`, `+`, `-`, `[`, `]`, `.`, `,`) operating on a 30,000-cell memory tape. All other characters are ignored as comments. Solving problems requires reasoning about pointer arithmetic, loop invariants, and memory layout without named variables, functions, or high-level control structures.

Befunge-98 (1993): Created by Chris Pressey as a two-dimensional stack-based language where the instruction pointer can travel in four cardinal directions. The `>`, `v`, `<`, and `^` commands set direction; conditional commands `_` and `|` branch based on stack values. The `p` (put) and `g` (get) commands enable self-modifying code.

Whitespace (2003): Created by Edwin Brady and Chris Morris. Only space, tab, and newline characters have semantic meaning; all other characters are ignored, meaning Whitespace programs

can be hidden within other text. The language is stack-based with commands encoded as whitespace sequences.

Unlambda (1999): Created by David Madore as a minimal functional language based on combinatory logic with no variables, only function application via the backtick character. Core combinators are `s` (substitution), `k` (constant), and `i` (identity). Even simple arithmetic requires constructing Church numerals through combinator compositions.

Shakespeare (2001): Created by Karl Wiberg and Jon Åslund. Programs are theatrical plays where variable declarations are character introductions, scenes/acts control program flow, and dialogue performs computation. Variable values are determined by adjectives: positive words ("beautiful", "fair") contribute positive values while negative ones ("damned", "evil") contribute negative values.

C Dataset Specification

Table 5: Distribution of problems across programming categories

Category	Count	Representative Problems
Basic I/O	5	Hello World, Echo Line
Arithmetic	17	Sum, Multiply, Factorial
String Manipulation	26	Reverse, Palindrome, Caesar
Number Theory	8	GCD, Primes, Factorization
Base Conversion	4	Binary↔Decimal
Sorting/Arrays	9	Sort, LIS, Inversions
Stack/Parsing	5	Balanced Parens, Postfix
State Machines	4	Tape Walk, Josephus
Bitwise Operations	2	Hamming, Count Bits

C.1 Sample Problems

Easy (E04): Sum Two Integers. Read two integers separated by whitespace, output their sum.

```
Input: "5 7" -> Output: "12"
Input: "-3 10" -> Output: "7"
```

Medium (M07): Factorial. Read N ($0 \leq N \leq 10$), output $N!$.

```
Input: "5" -> Output: "120"
Input: "0" -> Output: "1"
```

Hard (H03): Count Primes. Read N, count primes $\leq N$.

```
Input: "10" -> Output: "4"
Input: "100" -> Output: "25"
```

Extra-Hard (X20): Josephus Problem. Read N and K. N people in a circle, count K clockwise and eliminate. Output the survivor.

```
Input: "5 2" -> Output: "3"
Input: "7 3" -> Output: "4"
```

D Full Problem List

Easy (E01–E20): Hello World, Echo Line, Hello Name, Sum Two Integers, Multiply Two Integers, Even Or Odd, String Length, Reverse String, Count Vowels, Sum From 1 To N, Sum Of Digits, Minimum Of Two, Maximum Of Three, Repeat String N Times, Concatenate Two Lines, First And Last Character, Uppercase String, Count Spaces, Integer Average Of Two, Compare Two Integers.

Medium (M01–M20): Palindrome Check, Word Count, Run Length Encoding, Caesar Shift By 3, Simple Binary Expression, Greatest Common Divisor, Factorial, Nth Fibonacci Number, Decimal To Binary, Binary To Decimal, Substring Occurrences, Remove Vowels, Sort Numbers, Second Largest Distinct Number, Anagram Test, Interleave Two Strings, Replace Spaces With Underscores, Sum Of List, Characters At Even Indices, Count Distinct Characters.

Hard (H01–H20): Balanced Parentheses, Evaluate Expression With Precedence, Count Primes Up To N, Nth Prime Number, Big Integer Addition, Longest Word, Longest Common Prefix, Digit Frequency, General Caesar Cipher, Remove Consecutive Duplicates, Run Length Decoding, ASCII Sum, Polynomial Evaluation, List All Divisors, Tape Walk Final Position, Longest Run Length, Most Frequent Value, Divisible By 3, Plus Minus Reset Machine, Sort Strings Lexicographically.

Extra-Hard (X01–X20): Prime Factorization, Longest Increasing Subsequence Length, Matrix Multiplication Result Element, Evaluate Postfix Expression, Merge Two Sorted Arrays, Compute Power Modulo, Longest Palindromic Substring Length, Count Set Bits In Range, Bracket Depth Maximum, String Rotation Check, Count Inversions, Least Common Multiple, Valid Parentheses Types, Next Greater Element, Spiral Matrix Traversal, Hamming Distance, Roman To Integer, Integer To Roman, Permutation Check, Josephus Problem.

E Extended Results

Table 6: Per-language results (Easy solved / 20)

Model	0-Shot	Few	Self-S	ReAct	Best
			Brainfuck		
GPT-5.2	2	2	**5**	4	6.2%
O4-mini	2	3	**4**	3	5.0%
Gemini	2	3	**4**	3	5.0%
			Befunge-98		
GPT-5.2	2	7	**9**	7	11.2%
O4-mini	5	6	**8**	6	10.0%
Gemini	4	3	**6**	5	7.5%
			Whitespace, Unlambda, Shakespeare		
Best Model	0	0	0	0	0% (WS)
Best Model	0	1	1	0	1.2% (Unl)
Best Model	2	1	2	1	2.5% (Shk)

E.1 Agentic System Detailed Results

Table 7: Agentic system performance (Easy solved / 20)

System	Brainfuck	Befunge-98	Total (Acc)
Codex (OpenAI)	**11 (13.8%)**	7 (8.8%)	**18 (11.2%)**
Claude Code	10 (12.5%)	7 (8.8%)	17 (10.6%)

Codex - Brainfuck (11/20 Easy = 13.8%): Solved E01, E02, E03, E06, E07, E08, E09, E15, E16, E17, E18

Claude Code - Brainfuck (10/20 Easy = 12.5%): Solved E01, E02, E03, E08, E09, E14, E15, E16, E17, E18

Key failure pattern: Decimal I/O problems (E04, E05, E10-E13, E19, E20) account for most failures since parsing ASCII digits and multi-cell arithmetic have scarce training data.

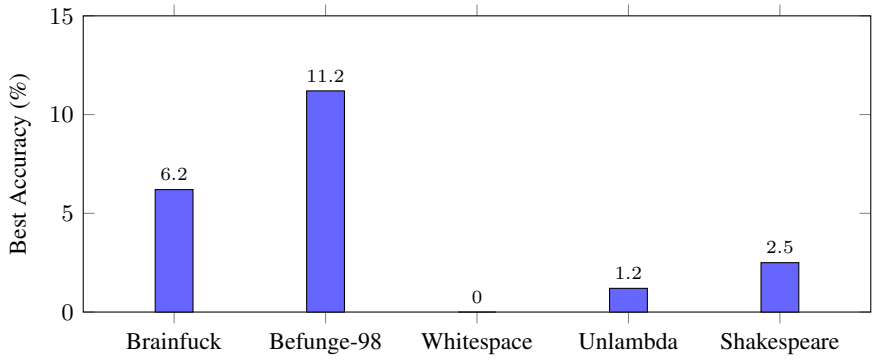

Figure 3: Best accuracy per language (all models/strategies). Befunge-98 most tractable (11.2%), Whitespace unsolved (0%).

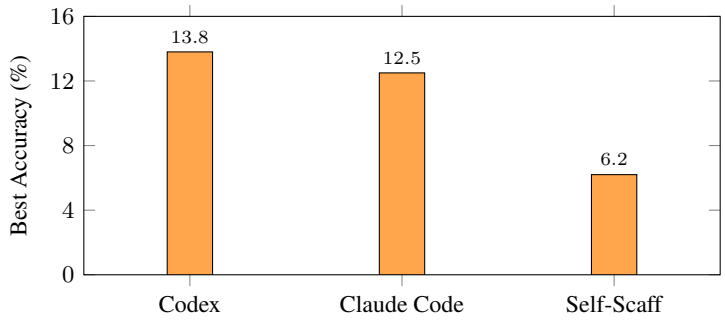

Figure 4: Agentic systems achieve 2× higher accuracy than best non-agentic (self-scaffolding) approach.

E.2 Performance Visualization

F Extended Error Analysis

Error profiles are consistent across models, suggesting language-specific limitations. Compile error rates: Whitespace 100%, Unlambda 88–95%, Befunge-98 20–30%, Brainfuck 12–20%. Logic error rates inversely correlate with compile errors, highest for Brainfuck (55–65%).

G Prompting Templates

Zero-Shot System:

```
You are an expert {language} programmer.
Output ONLY valid code. No explanations.
Reference: {documentation}
```

Self-Scaffolding (after iteration):

```
Previous: {code}
Feedback: Expected {expected}, Got {actual}
Error: {error_type}
Return updated program only.
```

Critic (Textual Self-Scaffolding):

```
Analyze this failing {language} program.
```

```
Explain issues in natural language only.
Do not write code.
```

ReAct Planner:

```
Analyze this problem and create a step-by-step
algorithm in pseudocode:
Problem: {problem_description}
Output a clear, numbered algorithm.
```

ReAct Code Editor:

```
Translate this algorithm to {language}:
Algorithm: {planner_output}
Documentation: {documentation}
Output only the program.
```

H Interpreter Specifications

All interpreters use consistent Python interfaces:

```
result = interpreter.run(
    code: str,
    stdin: str,
    timeout: float = 5.0
) -> ExecutionResult

@dataclass
class ExecutionResult:
    stdout: str
    stderr: str
    exit_code: int
    error_type: str  # ok, compile, runtime, timeout
```

Supported Languages:

- **Brainfuck**: 30,000-cell tape, 8-bit cells with wraparound
- **Befunge-98**: 2D grid with toroidal wrapping, 200,000 step limit
- **Whitespace**: Stack-based with heap, 28 instructions
- **Unlambda**: Functional with S, K, I combinators
- **Shakespeare**: Variable-per-character model with stack operations

I Agentic System Analysis

I.1 Case Study: Successful Problem (E01)

Problem: Print "Hello World!" with no input.

Attempt 1 (Success): Codex retrieves a canonical Hello World implementation and generates valid Brainfuck using standard idioms: initialize cells via multiplication loops, then output characters. Solved in 1 attempt with direct pattern retrieval.

I.2 Case Study: Failed Problem (E04)

Problem: Read two integers separated by whitespace, output their sum.

Attempts 1–10: Model assumes single-digit operands, fails to handle multi-digit numbers, and increasingly degenerates to constant outputs.

Analysis: Decimal I/O in Brainfuck requires: (1) parsing ASCII digits into numeric values, (2) handling variable-length numbers, (3) performing arithmetic on multi-cell representations, and (4) converting results back to ASCII. This pattern appears in $<0.1\%$ of Brainfuck programs online, making it effectively out-of-distribution.