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

Aman Sharma **Paras Chopra**
Lossfunk
{aman.sharma, paras}@lossfunk.com
Code: https://github.com/Lossfunk/EsolangBench
Dataset: https://huggingface.co/datasets/arcAman07/Esolang-Bench

ABSTRACT

Large language models achieve near-ceiling performance on code generation benchmarks, yet these results increasingly reflect memorization rather than genuine reasoning. We introduce **EsoLang-Bench**, a benchmark using five esoteric programming languages (Brainfuck, Befunge-98, Whitespace, Unlambda, and Shakespeare) that lack benchmark gaming incentives due to their economic irrationality for pre-training. These languages require the same computational primitives as mainstream programming but have 1,000–100,000$\times$ fewer public repositories than Python (based on GitHub search counts). We evaluate five frontier models across five prompting strategies and find a dramatic capability gap: models achieving 85–95% on standard benchmarks score only **0–11%** on equivalent esoteric tasks, with **0% accuracy beyond the Easy tier**. Few-shot learning and self-reflection fail to improve performance, suggesting these techniques exploit training priors rather than enabling genuine learning. EsoLang-Bench provides the first benchmark designed to mimic human learning by acquiring new languages through documentation, interpreter feedback, and iterative experimentation, measuring transferable reasoning skills resistant to data contamination.

1 INTRODUCTION

Large language models have achieved impressive performance on code generation benchmarks, with state-of-the-art systems reaching 85-95% accuracy on HumanEval and MBPP in mainstream programming languages (Chen et al., 2021; Austin et al., 2021). However, these 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).

We argue that evaluating models on truly *out-of-distribution* (OOD) tasks is essential for measuring genuine reasoning capabilities. Esoteric programming languages offer a principled solution: they have minimal representation in training corpora, making them economically irrational to include in pre-training data, while still requiring the same fundamental computational reasoning (loops, conditionals, state management) as mainstream languages.

We make the following key contributions:

1. We introduce **EsoLang-Bench**, a dataset of 80 programming problems spanning four difficulty levels, designed for evaluation across five esoteric programming languages with diverse computational paradigms.
2. We conduct comprehensive experiments evaluating five state-of-the-art LLMs across five prompting strategies, revealing that **in-context learning provides no significant benefit** for OOD tasks, confirming that ICL effectiveness depends on pre-training data coverage.

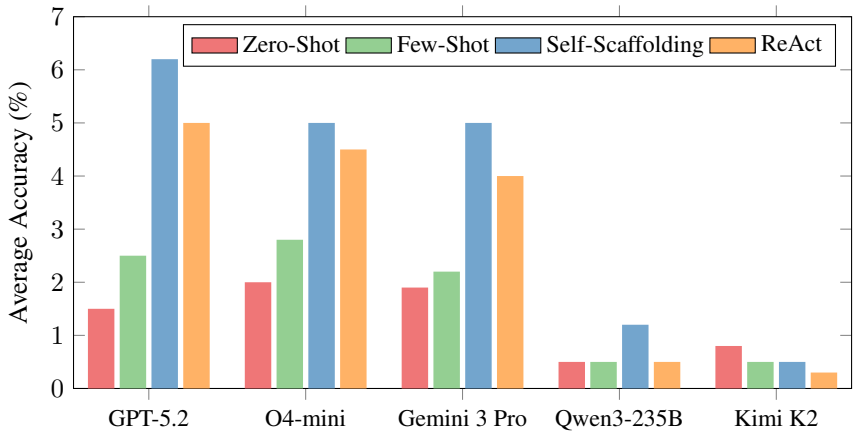

Figure 1: Average accuracy across all five esoteric languages by model and prompting strategy. Self-Scaffolding consistently achieves the highest accuracy, with GPT-5.2 reaching 6.2%. All models perform below 7% even with advanced scaffolding.

3. We benchmark agentic systems (Claude Code, Codex) with tool access, demonstrating that efficient context management and iterative feedback loops with interpreters are critical for OOD tasks.
4. We provide in-depth error analysis of benchmarking experiments, showing that compilation errors dominate (59%) even in self-scaffolding approaches, revealing fundamental syntactic gaps due to insufficient pre-training coverage.

2 Related Work

2.1 Code Generation Benchmarks

The evaluation of code generation capabilities has evolved through several generations of benchmarks. HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) established the paradigm of function-level synthesis with test-case verification. SWE-bench (Jimenez et al., 2024) extended this to repository-level tasks requiring understanding of complex codebases. MultiPL-E (Cassano et al., 2023) broadened language coverage to 18 programming languages, while DS-1000 (Lai et al., 2023) focused specifically on data science tasks. AlphaCode (Li et al., 2022) demonstrated competitive performance on programming competitions.

Specialized code models have achieved impressive results on these benchmarks. StarCoder (Li et al., 2023), Code Llama (Rozière et al., 2023), CodeGen (Nijkamp et al., 2023), InCoder (Fried et al., 2023), and CodeT5 (Wang et al., 2021b) represent the progression of code-specialized architectures. However, these benchmarks focus exclusively on mainstream languages with abundant training data, leaving open the question of whether high performance reflects genuine reasoning or pattern retrieval.

2.2 Benchmark Contamination and Gaming

The integrity of LLM evaluation has come under increasing scrutiny. Zhang et al. (2024) demonstrated systematic overperformance on GSM8k (Cobbe et al., 2021) compared to the contamination-free GSM1k variant, with some models exhibiting accuracy gaps of up to 8%. Sainz et al. (2023) found widespread contamination across NLP benchmarks. Deng et al. (2024) documented significant memorization affecting benchmark scores across model families, while Zhou et al. (2023) and Xu et al. (2024) quantified leakage patterns across benchmark families.

Most strikingly, Gupta et al. (2024) revealed that simply changing the order of multiple-choice answers can decrease MMLU accuracy by up to 13%, demonstrating that models exploit superficial patterns rather than understanding content. This phenomenon reflects Goodhart's Law (Goodhart,

1984): "when a measure becomes a target, it ceases to be a good measure." Strathern (1997) extended this to academia, and we argue it applies directly to AI benchmarks.

Jacovi et al. (2023) proposed practical strategies for mitigating contamination, while Oren et al. (2023) developed methods for proving contamination in black-box models. Bowman & Dahl (2021) argued for fundamental reforms to NLU benchmarking, and Raji et al. (2021) critiqued the proliferation of narrow benchmarks. Ribeiro et al. (2020) introduced behavioral testing methodologies that go beyond aggregate accuracy metrics.

2.3 Out-of-Distribution Generalization

Theoretical foundations for OOD generalization come from domain adaptation theory (Ben-David et al., 2010; Ganin et al., 2016), which establishes formal bounds on cross-domain transfer. Compositional generalization studies (Dziri et al., 2023) reveal systematic failures in transformer architectures when faced with novel combinations of known primitives. Anil et al. (2022) demonstrated failure of length generalization, while Merrill & Sabharwal (2024) analyzed the theoretical expressive power of transformers with chain-of-thought.

CrossFit (Ye et al., 2021) and Adversarial GLUE (Wang et al., 2021a) studied robustness to distribution shift in NLP. Chollet (2019) proposed measuring intelligence through skill acquisition efficiency rather than task-specific performance, while Mitchell (2021) surveyed abstraction and reasoning capabilities. We extend this line of work by proposing esoteric programming languages as controlled probes for OOD reasoning in code generation.

2.4 Advanced Prompting and Reasoning

In-context learning (Brown et al., 2020) demonstrated that large language models can adapt to new tasks through examples. Chain-of-thought prompting (Wei et al., 2022) and scratchpads (Nye et al., 2021) improved reasoning by eliciting intermediate steps. Self-improvement methods including Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and self-debugging (Chen et al., 2023) enable iterative refinement through self-generated feedback.

ReAct (Yao et al., 2023) synergizes reasoning with tool use, while program synthesis research (Gulwani et al., 2017; Ellis et al., 2021) provides theoretical foundations for inductive programming. Model-agnostic meta-learning (Finn et al., 2017) established frameworks for rapid adaptation. We systematically compare these approaches on OOD tasks, revealing that advanced prompting strategies may amplify rather than bridge knowledge gaps when foundational understanding is absent.

3 The EsoLang-Bench Dataset

3.1 Dataset Design

EsoLang-Bench consists of 80 programming problems organized into four difficulty levels with 20 problems each (see Figure 2). Each problem includes a natural language description and 6 input-output test cases for automated evaluation. Problems are designed to test fundamental computational reasoning rather than domain-specific knowledge: unlike HumanEval or MBPP which test familiarity with standard library functions, our problems require only basic algorithmic reasoning that transfers across programming paradigms. Problems are language-agnostic and can be implemented in any of the five target languages. The complete list of all 80 problems with full descriptions and test cases is provided in Appendix B, including representative sample problems from each difficulty tier.

Difficulty tiers are defined by algorithmic complexity of the Python reference solution, independent of observed model performance: **Easy** problems require single-loop or basic I/O (e.g., sum two integers, reverse a string); **Medium** problems require multi-loop control flow or basic recursion (e.g., Fibonacci, factorial); **Hard** problems require nested data structures or non-trivial string algorithms (e.g., balanced parentheses, prime counting); **Extra-Hard** problems require classical algorithms with complex state management (e.g., longest increasing subsequence, Josephus problem). Calibration was performed by expert assessment of Python solution complexity prior to any model evaluation.

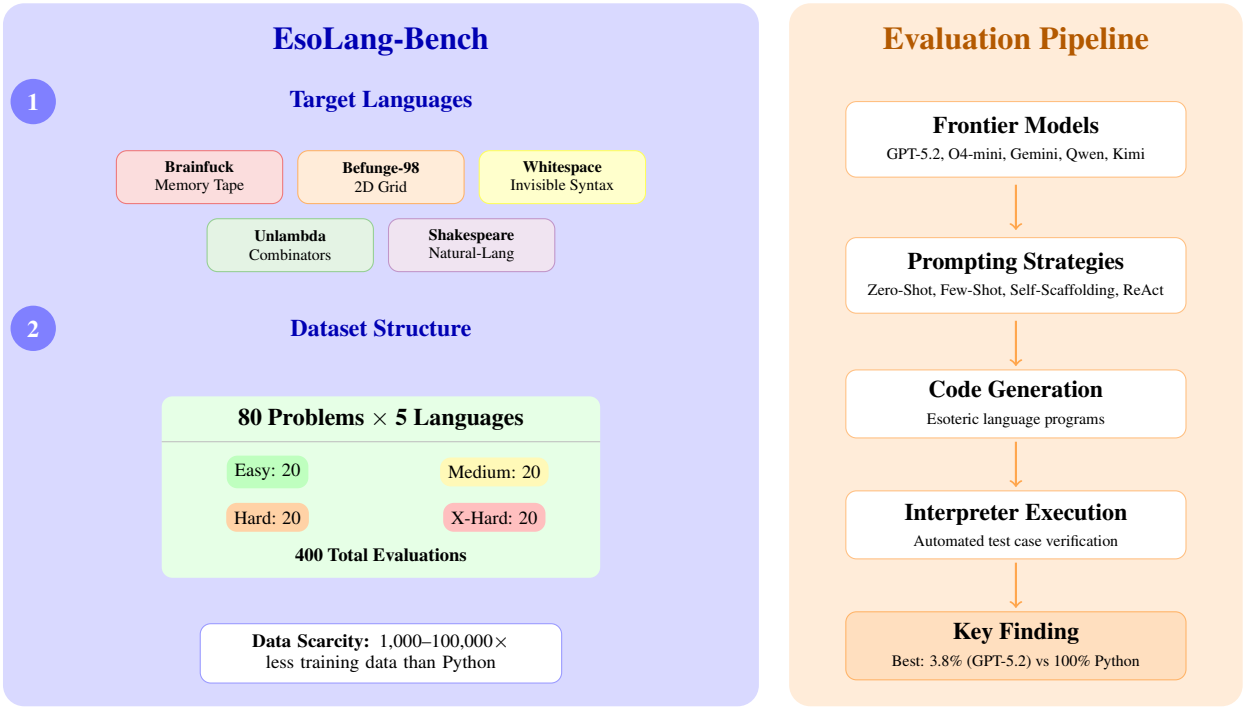

Figure 2: **EsoLang-Bench Overview.** *Left:* The benchmark comprises five esoteric programming languages spanning diverse computational paradigms, with 80 problems across four difficulty tiers (400 total evaluations). *Right:* Evaluation pipeline testing five frontier models across multiple prompting strategies, with automated interpreter-based verification. The best model achieves only 3.8% accuracy compared to 100% on equivalent Python problems.

3.2 Target Languages

We select five esoteric languages representing diverse computational paradigms:

Brainfuck: A memory-tape language with only 8 commands operating on a 30,000-cell tape. Requires pointer arithmetic and loop reasoning without variables or functions.

Befunge-98: A 2D stack-based language where the instruction pointer travels in four cardinal directions. Requires spatial reasoning and non-linear program flow.

Whitespace: Only space, tab, and newline characters have semantic meaning; all other characters are ignored. Stack-based with commands encoded as whitespace sequences.

Unlambda: A pure functional language based on combinatory logic with no variables; only function application via combinators (s, k, i).

Shakespeare: Programs are theatrical plays where dialogue performs computation. Natural-language-like syntax with entirely different semantics.

3.3 Data Scarcity as a Feature

The key advantage of esoteric languages is their extreme data scarcity. Figure 3 visualizes the dramatic gap: while Python appears in over 10 million repositories, esoteric languages have 1,000 to 100,000× fewer public GitHub repositories. This scarcity makes esoteric languages economically irrational to include in pre-training data: no deployment value justifies the training cost, data collection costs are high, and training would likely harm performance on mainstream programming tasks. Unlike traditional benchmarks where gaming provides competitive advantage, success on EsoLang-Bench can only come from genuine reasoning transfer.

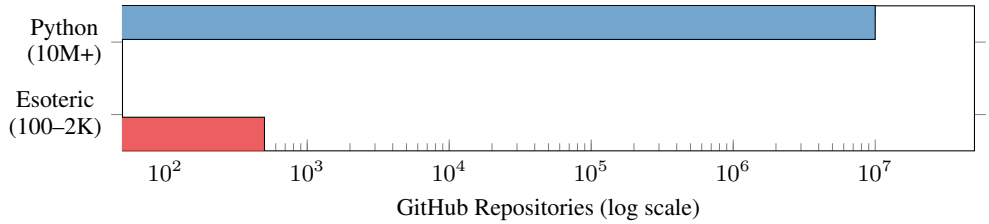

Figure 3: Training data scarcity (log scale). Esoteric languages have 5,000× fewer GitHub repositories than Python.

Table 1: Comparison of static vs. OOD benchmark paradigms across key evaluation dimensions.

ASPECT	STATIC	**OOD (OURS)**
CONTAMINATION RISK	HIGH	MINIMAL
GAMING INCENTIVE	HIGH	NONE
EVALUATES	RETRIEVAL	REASONING
TEST-TIME LEARNING	FORBIDDEN	ENABLED
INTERPRETABILITY	LIMITED	HIGH
HUMAN ALIGNMENT	WEAK	STRONG

3.4 LANGUAGE SELECTION CRITERIA

We selected these five esoteric languages based on four key properties that make them ideal for evaluating genuine reasoning capabilities:

(1) Turing Completeness: All five languages are Turing-complete, meaning they can express any computable function. This ensures that failure to solve problems reflects inability to reason about the language's computational model, not inherent language limitations.

(2) Paradigm Diversity: Each language represents a fundamentally different computational paradigm: memory-tape manipulation (Brainfuck), 2D spatial execution (Befunge-98), invisible syntax encoding (Whitespace), pure combinatory logic (Unlambda), and natural-language-like syntax with alien semantics (Shakespeare). Success requires transferable reasoning skills that generalize across paradigms, not memorization of language-specific patterns.

(3) Interpreter Availability: All languages have well-documented, open-source interpreters enabling automated evaluation with immediate execution feedback. This supports test-time learning experiments where models can iteratively refine solutions based on interpreter output.

(4) Data Scarcity: Public repositories are 1,000–100,000× scarcer than for mainstream languages (Figure 3), making contamination economically irrational while still providing enough examples for human experts to learn the languages. Full language specifications are provided in Appendix A.

Together, these criteria ensure that EsoLang-Bench measures genuine reasoning transfer rather than memorization, while remaining practically evaluable through automated testing. The combination of computational universality with extreme data scarcity creates an ideal testbed for distinguishing pattern matching from true algorithmic understanding.

4 BENCHMARK DESIGN PHILOSOPHY

EsoLang-Bench is designed around three core principles that address fundamental limitations of existing code generation benchmarks. Table 1 contrasts the properties of traditional static benchmarks with our proposed OOD evaluation paradigm.

4.1 THE BENCHMARK GAMING CYCLE

EsoLang-Bench breaks the benchmark gaming cycle by targeting domains where optimization is counterproductive. No rational actor would invest compute in esoteric language data that offers no deployment value, is costly to collect, and would degrade performance on profitable mainstream tasks. This *economic irrationality* is a structural guarantee against contamination that no post-hoc data-auditing method can fully provide.

4.2 MEASURING REASONING VS. RETRIEVAL

A fundamental question in LLM evaluation is whether high benchmark performance reflects genuine reasoning or sophisticated pattern retrieval. This distinction can only be assessed through out-of-distribution evaluation. If a model achieves 90% on Python tasks but 5% on equivalent Brainfuck tasks, the gap reveals the extent to which performance depends on training data rather than transferable computational reasoning.

Our benchmark design enables this comparison by constructing isomorphic problems across languages: the same algorithmic challenge (e.g., computing Fibonacci numbers) requires identical computational reasoning regardless of whether expressed in Python or Brainfuck. Performance differences thus isolate the contribution of language-specific training data from language-agnostic reasoning capability.

5 EXPERIMENTAL SETUP

5.1 MODELS EVALUATED

We evaluate five state-of-the-art LLMs representing the frontier of code generation capability: **GPT-5.2** (OpenAI), **O4-mini-high** (OpenAI reasoning model), **Gemini 3 Pro** (Google), **Qwen3-235B** (Alibaba), and **Kimi K2 Thinking** (Moonshot). All models were accessed via API to ensure consistent evaluation conditions.

5.2 PROMPTING STRATEGIES

We evaluate five prompting strategies with increasing complexity, designed to systematically probe how different forms of scaffolding interact with out-of-distribution tasks:

Zero-Shot: The model receives only the language documentation, problem description, and test case specifications. The system prompt establishes the model as an expert programmer and instructs it to output only valid code without explanations or markdown formatting. This baseline tests pure generalization without any task-specific examples.

Few-Shot: Extends zero-shot by prepending 3 solved example programs in the target esoteric language, demonstrating correct syntax and I/O patterns. Examples are selected to cover basic constructs (loops, I/O, arithmetic) without revealing solutions to evaluation problems. This tests whether in-context learning can bridge knowledge gaps.

Self-Scaffolding: An iterative approach where the model generates code, receives interpreter feedback (actual vs. expected output, error messages, stack traces), and refines its solution for up to 5 iterations. Crucially, no separate critic model is involved; the model must self-diagnose issues from raw interpreter output using a single LLM call per iteration. This isolates the benefit of execution feedback from explicit self-reflection.

Textual Self-Scaffolding: A two-agent iterative process requiring two LLM API calls per iteration: (1) the *coder* generates code given the problem and any prior feedback, (2) a separate *critic* agent analyzes the failing code and interpreter output to provide natural-language debugging guidance. The critic explicitly cannot write code; it can only diagnose issues and suggest improvements. This tests whether externalized textual critique helps on OOD tasks.

ReAct Pipeline: A three-stage approach inspired by Yao et al. (2023): (1) a *planner* model generates a high-level algorithm in pseudocode, (2) a *code editor* translates the plan into the target esoteric

Table 2: Zero-shot and few-shot (3 ICL examples) accuracy (%) across all models and languages. Each language contains 80 problems (20 per difficulty tier). Only Easy-tier problems were solved; all Medium/Hard/Extra-Hard = 0%. Best results per language in **bold**. 0-S = Zero-Shot, 3-S = Three-Shot Few-Shot.

	Brainfuck		**Befunge-98**		**Whitespace**		**Unlambda**		**Shakespeare**	
Model	0-S	3-S	0-S	3-S	0-S	3-S	0-S	3-S	0-S	3-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 3 Pro	2.5%	3.8%	5.0%	3.8%	0%	0%	0%	0%	1.2%	1.2%
Qwen-235B	2.5%	1.2%	0%	0%	0%	0%	0%	**1.2%**	0%	0%
Kimi K2	0%	0%	2.5%	1.2%	0%	0%	0%	0%	1.2%	1.2%

Table 3: Scaffolding strategy accuracy (%) across all models and languages. S-S = Self-Scaffolding (direct interpreter feedback, 1 LLM call); TSS = Textual Self-Scaffolding (coder-critic pair, 2 LLM calls); Re = ReAct pipeline. Each language contains 80 problems. Best results in **bold**.

	Brainfuck			**Befunge-98**			**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 3 Pro	**5.0%**	3.8%	3.8%	**7.5%**	6.2%	7.5%	0%	0%	0%	0%	0%	0%	**1.2%**	0%	0%
Qwen-235B	**2.5%**	1.2%	2.5%	0%	0%	0%	0%	0%	0%	**1.2%**	0%	0%	**1.2%**	0%	0%
Kimi K2	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	**1.2%**	0%	0%

language, and (3) a *critic* analyzes execution failures and feeds back to the planner. This tests whether decomposing reasoning and implementation helps when language-specific knowledge is missing.

5.3 Agentic Systems

We additionally evaluate two agentic coding systems with tool access to understand how interpreter feedback loops affect OOD performance. **GPT-5.2 Codex** operates via the OpenAI Codex API with direct interpreter access and iterative refinement. **Claude Code (Opus 4.5)** uses Claude Opus 4.5 with persistent context across problems and direct terminal access for interpreter execution. These systems differ primarily in their feedback loop architecture and context management capabilities.

5.4 Evaluation Protocol

We evaluate all models on the **complete 80-problem dataset** across all five esoteric languages, yielding 400 model-problem-language combinations per prompting strategy. To ensure statistical reliability, we conduct **three independent runs** (seeds 0, 1, 2) for each configuration, using temperature $\tau = 0.7$ to capture output variability. We report mean accuracy with 95% confidence intervals computed via bootstrap resampling ($n = 1000$ iterations).

For scaffolding methods, we allow up to 5 iterations per problem with a 10-second timeout per execution. A problem is considered solved if and only if all six test cases produce exact-match interpreter output (i.e., the interpreter's stdout matches the expected output exactly, character-for-character). We apply Bonferroni correction when comparing multiple models to control family-wise error rate at $\alpha = 0.05$. Statistical significance between prompting strategies is assessed using paired Wilcoxon signed-rank tests.

6 Results

Tables 2 and 3 present our main experimental results. Accuracy is computed as the percentage of problems solved out of 80 total problems per language (20 Easy + 20 Medium + 20 Hard + 20

Table 4: Agentic system accuracy (%) on Brainfuck and Befunge-98. Both systems have direct interpreter access with iterative feedback loops.

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

Extra-Hard). A critical finding is that **all models achieve 0% on Medium, Hard, and Extra-Hard problems across all configurations**; success is limited entirely to the Easy tier.

6.1 Static Prompting Performance

Table 2 reveals performance correlates with training data availability: Befunge-98 achieves the highest accuracy, followed by Brainfuck and Shakespeare. All models achieve 0% on Whitespace and near-zero on Unlambda, revealing a sharp capability boundary for languages with fewer than 200 GitHub repositories.

Within the Easy tier, top models solve a substantial fraction of problems: GPT-5.2 self-scaffolding solves 9/20 Easy Befunge-98 problems (45%) and 5/20 Easy Brainfuck problems (25%). The Codex agentic system solves 11/20 Easy Brainfuck problems (55%). This tier-level view reveals a sharper empirical story than aggregate accuracy suggests: models are not uniformly failing, but exhibit a hard performance cliff precisely at the boundary between single-loop pattern mapping (Easy) and multi-step algorithmic reasoning (Medium and above), where all models score 0%.

In-context learning provides negligible benefit. Few-shot prompting shows no statistically significant improvement over zero-shot (Wilcoxon $p = 0.505$, average change $+0.8$ percentage points). This finding aligns with Min et al. (2022), who demonstrated that in-context learning effectiveness depends heavily on the pre-training distribution: when the target domain lies outside the pre-training corpus, demonstration examples cannot bridge the knowledge gap. Our results provide empirical support for this hypothesis in the ultra-low-resource setting: the minimal performance separation between zero-shot and few-shot conditions suggests that, when target domains lie outside the pre-training corpus, demonstration examples may not compensate for missing foundational knowledge. For practitioners working on similarly ultra-low-resource OOD tasks, few-shot example curation appears to yield limited returns.

6.2 Scaffolding Strategy Performance

Table 3 presents results for three iterative scaffolding approaches. Self-scaffolding yields the best overall result: GPT-5.2 achieves 11.2% on Befunge-98 ($p < 0.01$ vs. zero-shot). Textual self-scaffolding achieves comparable but slightly lower results, and the ReAct pipeline shows particular strength on Befunge-98 for O4-mini.

Wilcoxon signed-rank tests reveal no statistically significant difference between self-scaffolding and textual self-scaffolding ($p = 0.803$), though **self-scaffolding performs best among all non-agentic strategies** while using half the compute (1 vs. 2 LLM calls per iteration). This suggests the benefit derives from direct interpreter feedback rather than the textual critique mechanism; for OOD tasks, concrete execution traces provide sharper learning signals than another model's interpretation of failures.

6.3 Agentic System Performance

We additionally evaluate two agentic systems with tool access on Brainfuck and Befunge-98 (Table 4).

Both agentic systems achieve 2–3$\times$ improvement over non-agentic approaches, with Codex reaching 13.8% on Brainfuck, the highest single-language result. We attribute this to direct interpreter access, persistent context, and execution traces unavailable through standard APIs, as well as effective context management: agents can dynamically retrieve language documentation and relevant reference material on demand through tool calls, avoiding the context dilution that limits fixed-prompt approaches.

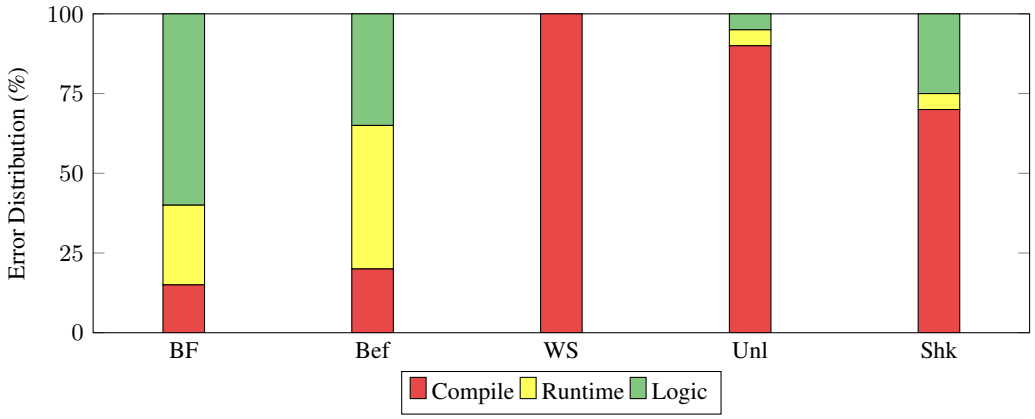

Figure 4: Error distribution by language (GPT-5.2 zero-shot). BF=Brainfuck, Bef=Befunge-98, WS=Whitespace, Unl=Unlambda, Shk=Shakespeare. Whitespace and Unlambda show near-total compile failure; Brainfuck shows primarily logic errors.

One counter-intuitive result warrants explanation: Codex agentic achieves 8.8% on Befunge-98, below the 11.2% achieved by GPT-5.2 self-scaffolding. This reflects both a model difference and an architectural one. First, Codex is a distinct model variant from GPT-5.2. Second, the agentic wrapper's context management and tool-call overhead (valuable for complex, multi-file tasks) reduces the effective number of rapid refinement iterations within the compute budget for short Befunge programs. For these compact programs, the direct 5-iteration feedback loop of self-scaffolding proves more efficient than the general-purpose agentic scaffolding.

7 Discussion

The experimental results reveal substantial limitations in current frontier models' ability to generalize computational reasoning to novel domains. The sharp difficulty boundary (0% accuracy on all problems beyond the Easy tier) suggests fundamental limitations rather than incremental capability gaps.

These results support the use of esoteric languages as ungameable benchmarks for evaluating genuine reasoning. Our language selection satisfies four key criteria: (1) Turing completeness: all five languages are computationally universal, ensuring problems are solvable in principle; (2) paradigm diversity: tape-based Brainfuck, stack-based 2D Befunge-98, whitespace-encoded Whitespace, functional combinatory Unlambda, and natural-language-style Shakespeare test different facets of transferable reasoning; (3) interpreter availability: deterministic, well-documented interpreters enable automated verification with immediate feedback; and (4) economic irrationality: minimal training data makes benchmark gaming counterproductive.

Error profiles reveal syntax vs. semantics gaps. Figure 4 shows the error distribution across languages. Languages with more online presence (Brainfuck, Befunge-98) show low compilation error rates (15–20%) but high logic error rates (35–60%), indicating models acquire surface syntax but fail on algorithmic reasoning. In contrast, ultra-low-resource languages (Whitespace, Unlambda) exhibit near-total compilation failure (90–100%), suggesting models cannot even generate valid syntax without sufficient training exposure. For Whitespace specifically, tokenizer behaviour may compound this effect (see Section 8). This binary pattern (syntax acquisition with semantic failure vs. complete syntactic failure) is consistent with a boundary between partial and absent pre-training coverage, though we note that additional confounds such as tokenizer behavior may contribute to failures in Whitespace (discussed in Section 8).

Few-shot learning provides negligible benefit in our ultra-low-resource setting. Prior work has shown that ICL effectiveness depends critically on pre-training data coverage (Min et al., 2022): demonstrations activate relevant pre-trained knowledge rather than teaching genuinely new skills. Our results provide empirical support for this hypothesis: the minimal performance separation between zero-shot and few-shot conditions (average $+0.8$ percentage points, $p = 0.505$) suggests that in

ultra-low-resource settings, demonstration examples may not compensate for absent foundational knowledge. This has practical implications: for similarly scarce-data domains, few-shot example curation appears to yield limited returns.

Direct interpreter feedback outperforms textual critique. Self-scaffolding (1 LLM call per iteration) matches or exceeds textual self-scaffolding (2 LLM calls per iteration) despite using half the compute. Direct interpreter feedback provides a sharper verification signal: concrete execution traces rather than another model's interpretation of failures. For OOD tasks where the critic also lacks domain knowledge, textual intermediaries introduce noise rather than signal. Detailed statistics are in Appendix C.

Agentic systems benefit from interpreter-in-the-loop architecture. Agentic coding systems (Codex, Claude Code) outperform non-agentic baselines through direct interpreter access, efficient context management that mitigates the *lost-in-the-middle* phenomenon (Liu et al., 2024), and task-specific example retrieval. Qualitative analysis in Appendix H shows Codex solves stream-processing problems in 1–3 attempts while decimal I/O problems consistently fail, revealing systematic capability boundaries.

8 LIMITATIONS

EsoLang-Bench has three main limitations. First, all success is concentrated in the Easy tier (0% on Medium and above), limiting differentiation among frontier systems at harder difficulties. Second, emerging techniques such as test-time reinforcement learning or retrieval-augmented generation are not evaluated. Third, our coarse error classification (compile, runtime, logic) may miss finer-grained failure patterns; for Whitespace in particular, near-total compilation failure may partly reflect BPE tokenizer normalisation of whitespace characters rather than reasoning limitations alone, which we leave for future work.

9 FUTURE WORK

We envision several directions for extending EsoLang-Bench as a community resource for measuring genuine reasoning capabilities:

Official Leaderboard. We plan to establish an official leaderboard for standardized evaluation and comparison of new models and methods. This will include held-out test sets to prevent overfitting to public examples and ensure fair comparison across submissions.

Language Expansion. A natural next step is extending EsoLang-Bench to additional esoteric languages (such as Malbolge, INTERCAL, and Piet) that represent qualitatively different failure modes and may expose further gaps in model reasoning. We will expand problem difficulty distributions and welcome community contributions of new languages and interpreters through a structured submission process.

Compute-Accuracy Analysis. A key future direction is systematic measurement of compute versus accuracy curves for esoteric language tasks, characterizing efficiency of different approaches and identifying whether additional compute yields meaningful improvements on OOD tasks.

10 CONCLUSION

We introduced EsoLang-Bench, a benchmark for evaluating LLM code generation on out-of-distribution tasks using esoteric programming languages. Our evaluation reveals that frontier models achieving near-perfect scores on standard benchmarks attain only 0–11% accuracy on equivalent esoteric tasks.

EsoLang-Bench establishes a first-principles benchmark for measuring transferable reasoning skills: the capacity to apply learned computational primitives to unfamiliar syntactic domains. Crucially, this benchmark mirrors how humans learn new programming languages: through documentation, interpreter feedback, and iterative experimentation rather than memorized examples. By targeting languages where benchmark gaming is economically irrational, our framework provides a robust, contamination-resistant evaluation that distinguishes genuine reasoning from memorization.

REFERENCES

Cem Anil, Yuhuai Wu, Anders Andreassen, Aitor Lewkowycz, Vedant Misra, Vinay Ramasesh, Ambrose Slone, Guy Gur-Ari, Ethan Dyer, and Behnam Neyshabur. Exploring length generalization in large language models. *Advances in Neural Information Processing Systems*, 2022.

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.

Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79:151–175, 2010.

Samuel R Bowman and George E Dahl. What will it take to fix benchmarking in natural language understanding? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4843–4855, 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.

Federico Cassano, John Gouwar, Daniel Lucchetti, Claire Schlesinger, Dennis Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 2023.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. Investigating data contamination in modern benchmarks for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 8706–8719, 2024. doi: 10.18653/v1/2024.naacl-long.482. URL https://aclanthology.org/2024.naacl-long.482/.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D Hwang, et al. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 2023.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850, 2021.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 1126–1135, 2017.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(59):1–35, 2016.

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

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

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

Alon Jacovi, Avi Caciularu, Omer Goldman, and Yoav Goldberg. Stop uploading test data in plain text: Practical strategies for mitigating data contamination by evaluation benchmarks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 5075–5084, 2023.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A natural and reliable benchmark for data science code generation. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 18319–18345, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 2023.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 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.

Melanie Mitchell. Abstraction and analogy-making in artificial intelligence. *Annals of the New York Academy of Sciences*, 1505(1):23–45, 2021.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Biber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B Hashimoto. Proving test set contamination in black box language models. *arXiv preprint arXiv:2310.17623*, 2023.

Inioluwa Deborah Raji, Emily M Bender, Amandalynne Paullada, Emily Denton, and Alex Hanna. AI and the everything in the whole wide world benchmark. *Advances in Neural Information Processing Systems*, 2021.

Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 4902–4912, 2020.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 2023.

Marilyn Strathern. 'improving ratings': Audit in the British university system. *European Review*, 5 (3):305–321, 1997.

Boxin Wang, Chejian Xu, Shuohang Wang, Zhe Gan, Yu Cheng, Jianfeng Gao, Ahmed Hassan Awadallah, and Bo Li. Adversarial GLUE: A multi-task benchmark for robustness evaluation of language models. In *Advances in Neural Information Processing Systems*, 2021a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, 2021b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

Cheng Xu, Shuhao Guan, Derek Greene, and M-Tahar Kechadi. Benchmark data contamination of large language models: A survey. *arXiv preprint arXiv:2406.04244*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Qinyuan Ye, Bill Yuchen Lin, and Xiang Ren. CrossFit: A few-shot learning challenge for cross-task generalization in NLP. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7163–7189, 2021.

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.

Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. Don't make your LLM an evaluation benchmark cheater. *arXiv preprint arXiv:2311.01964*, 2023.

REPRODUCIBILITY STATEMENT

We are committed to ensuring full reproducibility of our results. Upon publication, we will open-source: (1) the complete EsoLang-Bench dataset (80 problems, four difficulty tiers, 6 test cases each) in standardized JSON format; (2) Python interpreter implementations for all five esoteric languages with consistent interfaces for execution, timeout handling, and error classification; and (3) complete evaluation pipelines for all five prompting strategies, including prompt templates, API integration code, and automated scoring scripts. Raw experimental outputs for all model-language-strategy combinations are provided in the Appendix. All code and data will be released under an open-source license.

A 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.

B DATASET SPECIFICATION

B.1 DATASET STRUCTURE

EsoLang-Bench contains 80 problems organized into four difficulty tiers. Each problem includes:

- **ID**: Unique identifier (E01-E20, M01-M20, H01-H20, X01-X20)
- **Title**: Short descriptive name
- **Description**: Natural language specification
- **Test Cases**: 6 input-output pairs for automated verification

B.2 PROBLEM CATEGORY DISTRIBUTION

B.3 COMPLETE PROBLEM EXAMPLES

B.3.1 EASY: E04: SUM TWO INTEGERS

```
Title: Sum Two Integers
Description: Read two integers a and b separated by
```

Table 5: Distribution of problems across programming categories

Category	Count	Representative Problems
Basic I/O	5	Hello World, Echo Line, Concatenate Lines
Arithmetic	17	Sum, Multiply, Factorial, Modular Exponentiation
String Manipulation	26	Reverse, Palindrome, Caesar Cipher, RLE
Number Theory	8	GCD, Primes, Factorization, LCM
Base Conversion	4	Binary↔Decimal, Roman↔Integer
Sorting/Arrays	9	Sort, LIS, Inversions, Merge Sorted
Stack/Parsing	5	Balanced Parens, Postfix Eval, Expression Eval
State Machines	4	Tape Walk, Josephus Problem
Bitwise Operations	2	Hamming Distance, Count Set Bits

```
whitespace on one line. Output their sum a + b as a
plain integer with no extra text.

Test Cases:
1. Input: "5 7"      -> Output: "12"
2. Input: "-3 10"    -> Output: "7"
3. Input: "0 0"      -> Output: "0"
4. Input: "100 200"  -> Output: "300"
5. Input: "-50 -25"  -> Output: "-75"
6. Input: "999 1"    -> Output: "1000"
```

B.3.2 MEDIUM: M08: NTH FIBONACCI NUMBER

```
Title: Nth Fibonacci Number
Description: Read an integer N >= 1 and output the Nth
Fibonacci number using the 1-indexed sequence with
F1 = 1 and F2 = 1.

Test Cases:
1. Input: "1"  -> Output: "1"
2. Input: "5"  -> Output: "5"
3. Input: "10" -> Output: "55"
4. Input: "2"  -> Output: "1"
5. Input: "7"  -> Output: "13"
6. Input: "15" -> Output: "610"
```

B.3.3 HARD: H01: BALANCED PARENTHESES

```
Title: Balanced Parentheses
Description: Read a string made only of '(' and ')'
characters. Determine if the parentheses are balanced.
Output 'yes' if balanced, otherwise 'no'.

Test Cases:
1. Input: "()()"    -> Output: "yes"
2. Input: "((()))"  -> Output: "yes"
3. Input: "())("    -> Output: "no"
4. Input: "("       -> Output: "no"
5. Input: ""        -> Output: "yes"
6. Input: "(()())"  -> Output: "yes"
```

B.3.4 EXTRA-HARD: X20: JOSEPHUS PROBLEM

```
Title: Josephus Problem
Description: Read integers N and K. N people stand in
a circle numbered 1 to N. Starting from person 1,
count K people clockwise and eliminate that person.
Repeat until one remains. Output the survivor's number.

Test Cases:
1. Input: "5 2"  -> Output: "3"
2. Input: "7 3"  -> Output: "4"
3. Input: "1 1"  -> Output: "1"
4. Input: "6 1"  -> Output: "6"
5. Input: "10 2" -> Output: "5"
6. Input: "4 2"  -> Output: "1"
```

C EXTENDED RESULTS

C.1 LANGUAGE-SPECIFIC RESULTS BY BENCHMARK

Tables show Easy problems solved (out of 20 per language). Accuracy = Solved/80. All Medium/Hard/Extra-Hard = 0%.

Table 6: Brainfuck results by model and strategy (Easy solved / 20). Accuracy = Solved/80.

Model	0-Shot	Few	Self-S	ReAct	Best
GPT-5.2	2	2	**5**	4	**5 (6.2%)**
O4-mini	2	3	**4**	3	4 (5.0%)
Gemini 3 Pro	2	3	**4**	3	4 (5.0%)
Qwen3-235B	2	1	**2**	1	2 (2.5%)
Kimi K2	0	0	0	0	0 (0%)

Table 7: Befunge-98 results by model and strategy (Easy solved / 20). Accuracy = Solved/80.

Model	0-Shot	Few	Self-S	ReAct	Best
GPT-5.2	2	7	**9**	7	**9 (11.2%)**
O4-mini	5	6	**8**	6	8 (10.0%)
Gemini 3 Pro	4	3	**6**	5	6 (7.5%)
Qwen3-235B	0	0	0	0	0 (0%)
Kimi K2	2	1	2	0	2 (2.5%)

Table 8: Whitespace, Unlambda, Shakespeare results (Best Easy solved / 20). Accuracy = Solved/80.

Model	Whitespace	Unlambda	Shakespeare
GPT-5.2	0 (0%)	1 (1.2%)	2 (2.5%)
O4-mini	0 (0%)	0 (0%)	1 (1.2%)
Gemini 3 Pro	0 (0%)	0 (0%)	1 (1.2%)
Qwen3-235B	0 (0%)	1 (1.2%)	1 (1.2%)
Kimi K2	0 (0%)	0 (0%)	1 (1.2%)
Best	**0 (0%)**	**1 (1.2%)**	**2 (2.5%)**

Table 9: Agentic system performance (Easy solved / 20 per language)

System	Brainfuck	Befunge-98	Total (Acc)
Codex (OpenAI)	**11 (13.8%)**	7 (8.8%)	**18 (11.2%)**
Claude Code (Opus 4.5)	10 (12.5%)	7 (8.8%)	17 (10.6%)

C.2 Agentic System Detailed Results

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

Claude Code (Opus 4.5) - Brainfuck (10/20 Easy = 12.5%): E01, E02, E03, E08, E09, E14, E15, E16, E17, E18

Key failure pattern: Decimal number I/O (E04, E05, E10, E11, E12, E13, E19, E20) accounts for most Brainfuck failures across all systems.

C.3 Performance Visualization

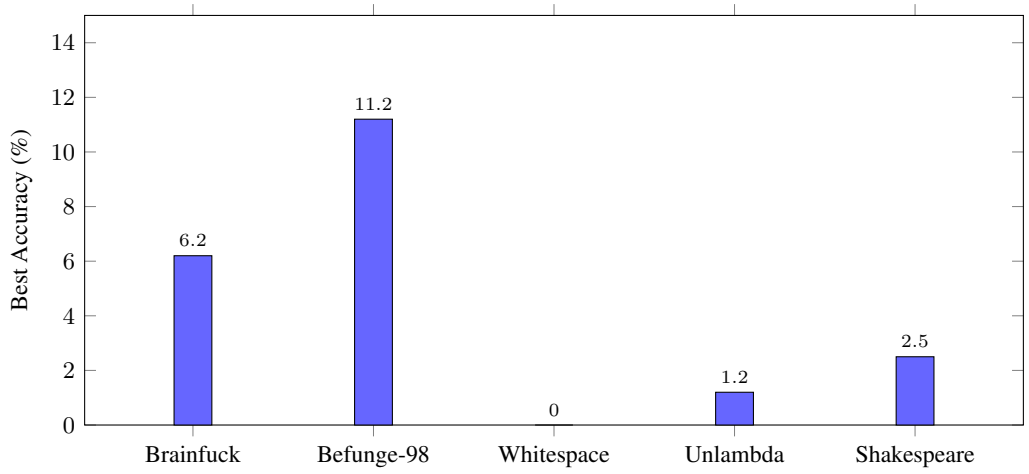

Figure 5: Best accuracy achieved per language (across all models and strategies). Befunge-98 is the most tractable (11.2%), while Whitespace remains completely unsolved (0%).

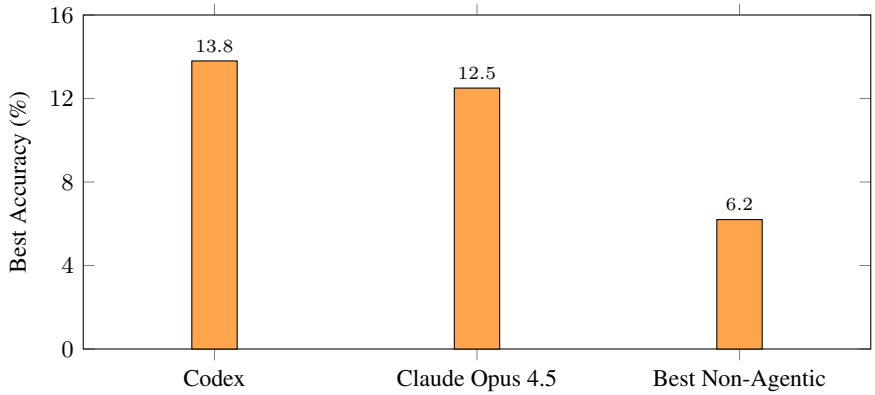

Figure 6: Agentic systems vs best non-agentic approach. Agentic coding systems achieve 2× higher accuracy than the best self-scaffolding approach.

D PROMPTING TEMPLATES

This section provides the exact prompts used for each strategy. Variables in {braces} are filled at runtime.

D.1 ZERO-SHOT PROMPT

System Prompt:

```
You are an expert {language_name} programmer. Given a
problem and sample tests, output ONLY valid code in
this esoteric language. No explanations, no comments,
no markdown. Programs must read stdin exactly as
specified and write deterministic stdout that matches
the expected output byte-for-byte.

Reference documentation:
{documentation_text}
```

User Prompt:

```
Problem ID: {problem_id}
Title: {problem_title}
Description:
{problem_description}

Specification tests (stdin => stdout):
1.
Input:
{test_1_input}
Output:
{test_1_output}

2.
Input:
{test_2_input}
Output:
{test_2_output}

[... additional test cases ...]

Return only the program.
```

D.2 FEW-SHOT PROMPT

Extends zero-shot by adding to the system prompt:

```
Here are solved examples for reference.
```

And prepending reference examples before the problem:

```
Example 1: {example_title}
Task: {example_description}
Program:
{example_program}
Sample I/O:
- Input: {io_1_input}
  Output: {io_1_output}
```

```
Example 2: {example_title}
[... 3 examples total ...]
```

D.3 Self-Scaffolding Prompt

System Prompt (extends zero-shot):

```
[Zero-shot system prompt]
Iteratively update your program using the prior code
and interpreter feedback provided.
```

User Prompt (after first attempt):

```
[Problem specification]

Previous attempt and interpreter feedback:
=== Program ===
{previous_code}

=== Interpreter Feedback ===
Test 1:
Input: {test_input}
Expected: {expected_output}
Actual: {actual_output}
Error Type: {error_type}
Stderr: {stderr}

[... additional test results ...]

Return only the updated program.
```

D.4 Textual Self-Scaffolding Prompt

Uses two separate prompts for coder and critic roles.

Coder System Prompt:

```
[Zero-shot system prompt]
Iteratively improve your solution when feedback
is provided.
```

Coder User Prompt (with feedback):

```
[Problem specification]

Previous attempt:
{previous_code}

Critic feedback:
{critic_feedback}

Return only the updated program.
```

Critic System Prompt:

```
You are an expert {language_name} reviewer. Analyse
the failing program and interpreter feedback. Explain
issues and suggest improvements in natural language
only. Do not write code.
```

Critic User Prompt:

```
Problem ID: {problem_id}
Title: {problem_title}
Description:
{problem_description}

Attempt details:
=== Program ===
{code}

=== Interpreter Feedback ===
[Test results with expected/actual outputs]

Provide concise debugging guidance without including
any code.
```

D.5 REACT PIPELINE PROMPT

Planner Prompt:

```
Analyze this problem and create a step-by-step
algorithm in pseudocode:

Problem: {problem_description}
Test Cases: {test_cases}

Output a clear, numbered algorithm that can be
translated to any programming language.
```

Code Editor Prompt:

```
Translate this algorithm to {language_name}:

Algorithm:
{planner_output}

Language Documentation:
{documentation}

Output only the program, no explanations.
```

Critic Prompt:

```
The {language_name} program produced incorrect output.

Expected: {expected_output}
Actual: {actual_output}
Error: {error_type}

Analyze the discrepancy and suggest specific changes
to the algorithm or implementation.
```

E INTERPRETER SPECIFICATIONS

All interpreters are implemented in Python with consistent interfaces:

```
result = interpreter.run(
    code: str,
    stdin: str = None,
    timeout_seconds: float = 5.0
```

```
) -> ExecutionResult

@dataclass
class ExecutionResult:
    stdout: str           # Program output
    stderr: str           # Error messages
    exit_code: int        # 0 = success
    error_type: str       # "ok", "compile_error",
                          # "runtime_error", "timeout"
```

E.1 Supported Languages

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

F Full Problem List

F.1 Easy Problems (E01-E20)

E01 Print Hello World
E02 Echo Line
E03 Hello Name
E04 Sum Two Integers
E05 Multiply Two Integers
E06 Even Or Odd
E07 String Length
E08 Reverse String
E09 Count Vowels
E10 Sum From 1 To N
E11 Sum Of Digits
E12 Minimum Of Two
E13 Maximum Of Three
E14 Repeat String N Times
E15 Concatenate Two Lines
E16 First And Last Character
E17 Uppercase String
E18 Count Spaces
E19 Integer Average Of Two
E20 Compare Two Integers

F.2 Medium Problems (M01-M20)

M01 Palindrome Check
M02 Word Count
M03 Run Length Encoding
M04 Caesar Shift By 3

M05 Simple Binary Expression
M06 Greatest Common Divisor
M07 Factorial
M08 Nth Fibonacci Number
M09 Decimal To Binary
M10 Binary To Decimal
M11 Substring Occurrences
M12 Remove Vowels
M13 Sort Numbers
M14 Second Largest Distinct Number
M15 Anagram Test
M16 Interleave Two Strings
M17 Replace Spaces With Underscores
M18 Sum Of List
M19 Characters At Even Indices
M20 Count Distinct Characters

F.3 HARD PROBLEMS (H01-H20)

H01 Balanced Parentheses
H02 Evaluate Expression With Precedence
H03 Count Primes Up To N
H04 Nth Prime Number
H05 Big Integer Addition
H06 Longest Word
H07 Longest Common Prefix
H08 Digit Frequency
H09 General Caesar Cipher
H10 Remove Consecutive Duplicates
H11 Run Length Decoding
H12 ASCII Sum
H13 Polynomial Evaluation
H14 List All Divisors
H15 Tape Walk Final Position
H16 Longest Run Length
H17 Most Frequent Value
H18 Divisible By 3
H19 Plus Minus Reset Machine
H20 Sort Strings Lexicographically

F.4 Extra-Hard Problems (X01-X20)

X01 Prime Factorization
X02 Longest Increasing Subsequence Length
X03 Matrix Multiplication Result Element
X04 Evaluate Postfix Expression
X05 Merge Two Sorted Arrays
X06 Compute Power Modulo
X07 Longest Palindromic Substring Length
X08 Count Set Bits In Range
X09 Bracket Depth Maximum
X10 String Rotation Check
X11 Count Inversions
X12 Least Common Multiple
X13 Valid Parentheses Types
X14 Next Greater Element
X15 Spiral Matrix Traversal
X16 Hamming Distance
X17 Roman To Integer
X18 Integer To Roman
X19 Permutation Check
X20 Josephus Problem

G Extended Error Analysis

This section provides detailed error analysis across all evaluated models. Figure 7 shows error distributions for O4-mini, Gemini 3 Pro, and Qwen3-235B under zero-shot conditions.

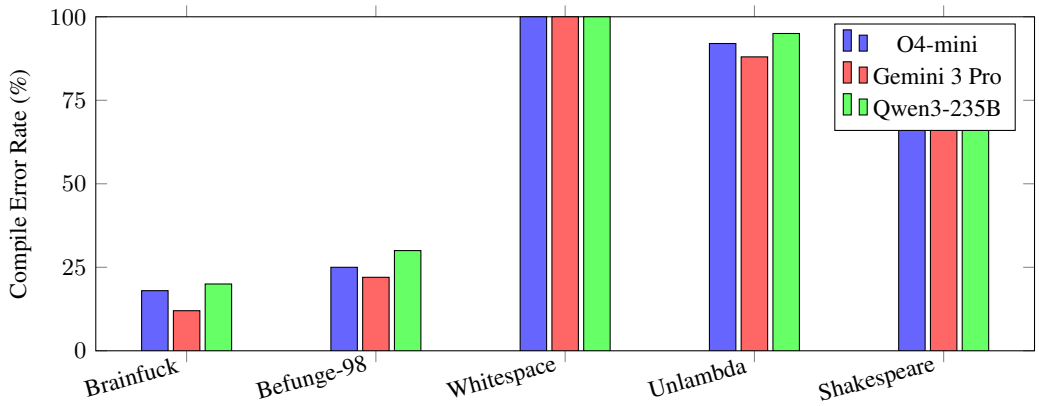

Figure 7: Compile error rates by model across languages. All models show near-identical patterns: low compile errors on Brainfuck/Befunge-98, complete failure (100%) on Whitespace, and high failure (88–95%) on Unlambda.

Key observations: (1) Error profiles are remarkably consistent across models, suggesting language-specific rather than model-specific limitations. (2) Whitespace achieves 100% compile errors across all models; no model can generate valid whitespace-only syntax. (3) Brainfuck shows the lowest compile error rates (12–20%) but highest logic error rates (55–65%), indicating models can learn syntax but not semantics.

H AGENTIC SYSTEM QUALITATIVE ANALYSIS

This section provides in-depth qualitative analysis of GPT-5.2 Codex performance on Brainfuck, based on detailed execution logs from our evaluation.

H.1 ARCHITECTURE OVERVIEW

Codex operates with an interpreter-in-the-loop architecture comprising: (1) direct interpreter access with 5 maximum attempts per problem, (2) structured logging that fetches relevant prior attempts to avoid context degradation, (3) task-family routing that retrieves semantically similar solved examples, and (4) efficient context management that mitigates the *lost-in-the-middle* phenomenon by selectively surfacing relevant information rather than maintaining full conversation history.

H.2 PERFORMANCE BY PROBLEM CATEGORY

Table 10 shows Codex performance stratified by problem category. Stream-processing problems (character I/O without numeric parsing) are solved reliably, while decimal I/O problems consistently fail.

Table 10: Codex Brainfuck results by problem category. Stream = character-based I/O; Decimal = requires numeric parsing/printing.

Problem	**Category**	**Solved**	**Attempts**
E01: Hello World	Stream	✓	1
E02: Echo Line	Stream	✓	1
E03: Hello Name	Stream	✓	1
E04: Sum Two Integers	Decimal	×	10
E05: Multiply Two Integers	Decimal	×	10
E06: Even Or Odd	Classify	✓	2
E07: String Length	Stream	✓	3
E08: Reverse String	Stream	✓	1
E09: Count Vowels	Classify	✓	3
E10: Sum 1 to N	Decimal	×	10
E15: Concatenate Lines	Stream	✓	2
E16: First/Last Char	Stream	✓	2
E17: Uppercase String	Stream	✓	2
E18: Count Spaces	Stream	✓	2

H.3 CASE STUDY: SUCCESSFUL PROBLEM (E01)

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

Attempt 1 (Success): Codex retrieves a canonical Hello World implementation from its ICL examples and generates:

```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.<<+++++++++++++++.
>.+++.------.--------.>+.
```

This solution uses standard Brainfuck idioms: initialize cells via multiplication loops, then output characters. Solved in 1 attempt with direct pattern retrieval.

H.4 CASE STUDY: FAILED PROBLEM (E04)

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

Attempt 1: Model assumes single-digit operands, produces incorrect output "¡" for input "5 7" (expected "12").

Attempt 2: Attempts to split digits into cells but fails to handle multi-digit numbers.

Attempts 3–10: Model increasingly degenerates, trying constant outputs ("12", "0", "=") and echo strategies that cannot generalize.

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 "decimal toolkit" pattern appears in $<0.1\%$ of Brainfuck programs online, making it effectively out-of-distribution even for the highest-resource esoteric language.

H.5 Architectural Insights

The interpreter-in-the-loop architecture provides three key advantages for OOD tasks:

1. Sharp feedback signal. Direct execution output ("actual: ¡, expected: 12") provides unambiguous error signal, unlike textual critique which may misdiagnose issues in unfamiliar domains.

2. Context efficiency. By logging attempts as structured JSON and fetching only relevant prior attempts, Codex avoids the attention dilution that occurs when LLMs must attend to long conversation histories.

3. Task-family retrieval. Routing problems to semantic categories (stream, classify, arithmetic) and retrieving category-specific examples outperforms generic few-shot demonstrations.

However, these advantages cannot overcome fundamental capability gaps: when the required algorithmic pattern (e.g., decimal parsing) is absent from pre-training, no amount of iteration or retrieval can synthesize it from first principles.