PBEBench: A Multi-Step Programming by Examples Reasoning Benchmark inspired by Historical Linguistics

Atharva Naik[1], Darsh Agrawal[1], Prakam[3], Yash Mathur[3], Manav Kapadnis[1], Yuwei An[1], Clayton Marr[2], Carolyn Rose[1], David Mortensen[1]

[1]Carnegie Mellon University, [2]Ohio State University, [3]Independent Researcher
{arnaik,darsha,mkapadni,yuweia,cprose,dmortens}@cs.cmu.edu

Abstract

While many benchmarks evaluate the reasoning abilities of Large Language Models (LLMs), few isolate reasoning as a capability independent of domain knowledge. We introduce a new benchmark for inductive reasoning inspired by Sound Law Induction (SLI) in historical linguistics and formulated in a simple multi-step Programming by Example (PBE) framework. The task requires inducing a cascade of string rewrite programs that transform inputs into target outputs. We present PBEBench, a fully automated evaluation approach that generates such problems with controllable difficulty and ordering constraints, enabling scalable and contamination-resistant evaluation of sequential inductive reasoning. Using this approach, we construct three datasets that show a large gap between models that leverage test-time compute or long chain-of-thought reasoning and those that do not. Although recent models such as GPT-5 and gpt-oss-120b show promise, solve rates remain below 5% on hard *PBEBench* instances with long program cascades, even under computationally expensive scaling strategies. Finally, we show that *PBEBench* scores are more predictive of performance on real SLI than are other inductive reasoning benchmarks. We release code and data to support further research.[1]

1 Introduction

In historical linguistics, explaining how ancestral word forms evolve into their modern reflexes requires positing an ordered sequence of sound changes. The order is critical: applying a change too early or too late can eliminate the conditions that enable other changes, producing incorrect outcomes. For example, if the change mapping Proto-Tangkhulic /i/ to /ɯ/ word-finally in Tusom had preceded the deletion of word-final /ŋ/, the Tusom word for 'snail' (from *liŋ) would be /li/ rather than the attested /lɯ/ Linguists say that the second rule FEEDS the first because it creates contexts where the first can apply. There are three other ordering-sensitive relations between sound changes: BLEEDING (A removes contexts for B), COUNTER-FEEDING (B would feed A if ordered first), and COUNTER-BLEEDING (B would bleed A if ordered first). Inferring such interactions has long been recognized as a core challenge in phonological theory (Kiparsky, 1968, 1971). At an abstract level, Sound Law Induction (SLI) can be viewed as a programming-by-example task: given input–output string pairs, the goal is to infer a sequence of local rewrite rules whose composition maps inputs to outputs (Naik et al., 2024, 2025b). Similar ordering constraints arise in other domains, such as multi-file code refactoring (Gautam et al.), where transformations must be applied in a consistent order while reasoning over evolving intermediate states under partial observability. Formally, given inputs $\vec{\imath}$ (ancestral words or input files) and corresponding outputs $\vec{o}$, the task is to infer any valid program *cascade* $\vec{p}$, of incremental transformations that maps $\vec{\imath}$ to $\vec{o}$.

(Naik et al., 2025b, 2024) show that LLMs struggle with SLI, but benefit more from training on logically consistent synthetic sound laws than on real sound laws from *Index Diachronica*. However, these works do not explain why this is the case. We hypothesize that the difficulty arises not from individual rewrite operations, but from inducing sequences of interacting transformations whose correctness depends on precise ordering and unobserved intermediate states, explaining the advantage of structurally consistent synthetic data. Despite minimal formal machinery, the problem exhibits rich interaction structure, making it a natural testbed for multi-step inductive reasoning and sequential planning in a domain-light setting (Ma et al., 2024). Moreover, while LLMs may have encountered string rewriting tasks during pre-

[1]https://github.com/cmu-llab/PBEBench

Findings of the Association for Computational Linguistics: ACL 2026, pages 8877–8918

training, PBEBench instances are randomly generated with controlled structure, making it unlikely that pre-training memorization or distributional biases significantly skew performance. We propose PBEBench, an automated approach towards generating such data automatically and scalably. PBEBench exhibits several desirable properties missing in existing reasoning benchmarks: (1) it does not rely on specialized background knowledge (see Section 2); (2) its difficulty emerges from a simple, formally specified, finite, problem space of sequential string rewrites; (3) it supports explicit control of fundamental interaction patterns—feeding, bleeding, counter-feeding, and counter-bleeding—studied in historical phonology (Kiparsky, 1968, 1971) (termed as BFCC relations here); (4) it is inherently resistant to data contamination and saturation, as new instances with controlled difficulty can be generated scalably; and (5) it evaluates models on practically and scientifically meaningful induction problems rather than human–model comparisons.

Our study investigates the following hypotheses:
H1: LLMs struggle with multi-step inductive reasoning. We benchmark 13 reasoning and 8 non-reasoning models on three PBEBench datasets to study reasoning bottlenecks in LLMs. The domain-light setting suggests that reasoning models should outperform non-reasoning ones.
H2: Difficulty in our dataset is driven by interaction structure. We hypothesize that while inducing a single string rewrite is straightforward, LLMs struggle to compose long cascades, especially under complex BFCC ordering constraints. We test this by contrasting increased per-step complexity (via more examples) with longer cascades and richer BFCC interactions.
H3: Performance on PBEBench predicts performance on real SLI. We evaluate all open-source LLMs on real SLI data and contemporary synthetic inductive reasoning benchmarks, and compare correlations to identify which benchmarks best predict real SLI performance.
H4: Scaling the thinking budget is more effective than increasing the sampling budget. We compare two scaling strategies: increasing the thinking budget (maximum sequence length) and the sampling budget, as described in Section 3.2, to determine which yields more "bang for the buck" under a fixed compute budget.

2 Related Work

Programming By Example. Programming by Example (PBE) (Gulwani, 2010) is a program synthesis paradigm where programs are inferred from input–output pairs. Early symbolic approaches rely on domain-specific languages and constraints: FlashFill uses a string-transformation DSL (Gulwani, 2011), while Syntax-Guided Synthesis (SyGuS) restricts program search via grammars (Alur et al., 2013). DeepCoder (Balog et al., 2017) learns function predictions to guide search, and RobustFill (Devlin et al., 2017) trains sequence-to-sequence models to directly emit DSL programs. More recently, Large Language Models show few-shot capability in code generation (Chen et al., 2021a; Guo et al., 2024) and test case generation (Li and Yuan, 2024), but on traditional PBE, they struggle with out-of-distribution data and typically improve after fine-tuning on the target distribution (Li and Ellis, 2024; Naik et al., 2025b).

Inducing Context-Sensitive Grammars in LLMs. Attempts to induce string-rewrite rules from data have a long history (Gildea and Jurafsky, 1995), with linguistic analyses of their formal properties dating back further Kiparsky (1968, 1971). Naik et al. (2024) show that LLMs can induce sound laws, extending this to full context-sensitive program synthesis (Naik et al., 2025b). However, these works do not provide provably correct algorithms for detecting feeding and bleeding relations, which is a unique contribution of this work (see Appendix B).

Reasoning and Induction Benchmarks. Several benchmarks have been proposed to evaluate reasoning and PBE capabilities. Code-centric suites such as HumanEval and MBPP assess function generation (Chen et al., 2021a; Austin et al., 2021), while FlashFill-style datasets target example-driven string transformations (Gulwani, 2011). More recent benchmarks emphasize software-engineering workflows (Jain et al., 2025; Zhang et al., 2025; Shao et al., 2025). However, software engineering tasks often entangle general reasoning with domain-specific knowledge (of tools and libraries), leading to uneven prior exposure and complicating contamination-free evaluation. Beyond code, multi-step, system-2, and mathematical reasoning are explored by HotpotQA, DROP, BIG-Bench Hard, and GSM8K (Yang et al., 2018; Dua et al., 2019; Suzgun et al., 2022; Cobbe et al., 2021). Inductive rule learning and compo-

sitional generalization are probed via CLUTRR and SLR (Sinha et al., 2019; Helff et al., 2025), bAbI and KOR-Bench (Weston et al., 2015; Ma et al., 2024), and extrapolative splits such as SCAN and CFQ (Lake and Baroni, 2018; Keysers et al., 2020); ARC provides a non-language analog (Chollet, 2019). In contrast, PBEBench targets multi-step synthesis of *string-rewriting cascades*, is easily scalable and leakage-resistant, and supports difficulty control through simple generation parameters.

3 Methodology

Our approach comprises two components: (1) a problem proposer and (2) a problem solver (Fig. 1a). The proposer is a symbolic system that generates PBE instances with controllable difficulty and forms the core of our dynamic benchmarking framework. It scalably produces novel, contamination-free input, output, and program triples of controllable difficulty. The solver corresponds to any system under evaluation. We benchmark state-of-the-art LLMs on inductive reasoning using a program reordering, multi-step PBE task, and additionally evaluate them on human-curated, low-resource Sound Law Induction (SLI) data.

3.1 Problem Proposer

The symbolic proposer (Fig 1a, top) constructs data for the multi-step PBE task by sampling inputs from a distribution of strings $\mathcal{I}$ and a sequence of string rewrite programs from a distribution $\mathcal{P}$. Each program acts as a find-and-replace function, substituting all occurrences of a substring α with a substring β (details in Section 3.1.2). The proposer applies the sampled *cascade* of programs $\vec{p} = \langle p_1, \ldots, p_m \rangle \in \mathcal{P}$ to the sampled inputs $\vec{i} = \langle i_1, \ldots, i_n \rangle \in \mathcal{I}$, producing outputs $\vec{o} = \langle p_m(\ldots p_1(i_1)), \ldots, p_m(\ldots p_1(i_n)) \rangle$. For convenience, we write this transformation as $\vec{o} = \vec{p}(\vec{i})$. Each dataset instance (Fig 1a, second from top) is defined by the triplet $\langle \vec{i}, \vec{p}, \vec{o} \rangle$, consisting of the inputs, the program cascade, and the corresponding outputs. In addition, the proposer records metadata describing interactions between programs (e.g., BLEEDING, FEEDING), illustrated in Fig 1b.

The proposer is parameterized by the number of examples n ($|\vec{i}| = |\vec{o}| = n$; $n = 5$ in Fig 1b); the input alphabet Σ, used to generate both inputs and string rewrite programs; the minimum and maximum input string lengths l_{min} and l_{max} ($l_{min} = 2$, $l_{max} = 6$ in Fig 1b); the minimum and maximum cascade lengths L_{min} and L_{max} ($L_{min} \leq |\vec{p}| \leq L_{max}$; $|\vec{p}| = 4$, $L_{min} = 2$, $L_{max} = 5$ in Fig 1b); the minimum and maximum substring lengths s_{min} and s_{max} for α and β in the rewrite programs ($s_{min} \leq |\alpha|, |\beta| \leq s_{max}$); and the desired dataset size D. We denote a benchmark snapshot generated with fixed parameter values as $\mathcal{D}(n, \Sigma, L_{min}, L_{max}, l_{min}, l_{max}, s_{min}, s_{max}, D)$.

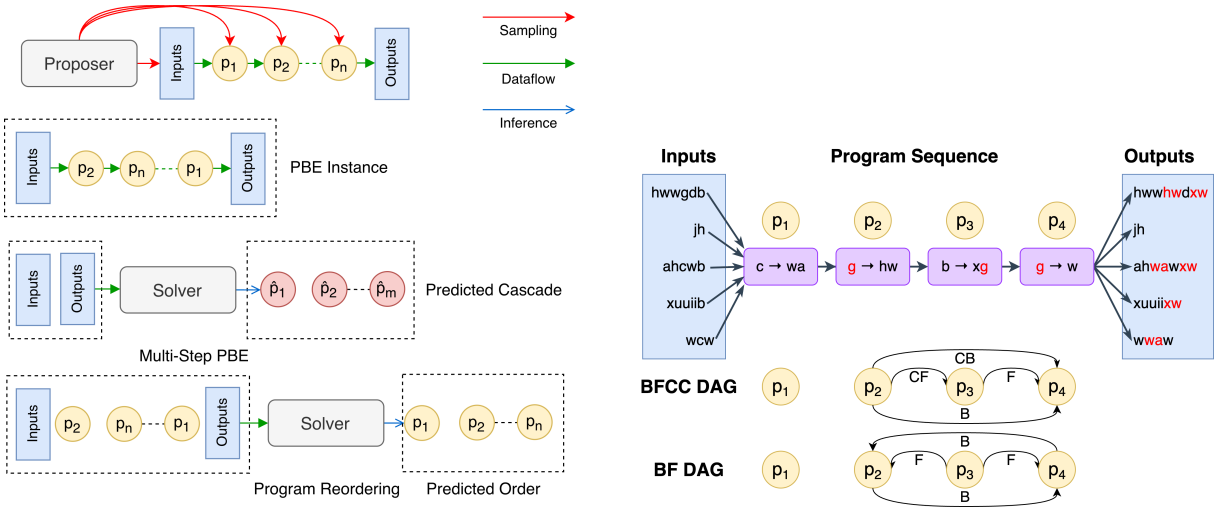
For the program reordering task, we take PBE instances (Fig 1a, second from top) generated by the proposer and permute the program sequence by swapping program pairs that exhibit FEEDING or BLEEDING interactions. This produces a permuted sequence $\sigma(\vec{p})$ and a different output vector $\vec{o}_\sigma = \sigma(\vec{p})(\vec{i}) \neq \vec{o}$. The goal of the LLM is to recover the original program sequence $\vec{p}$ from the original inputs $\vec{i}$, original outputs $\vec{o}$, and the permuted program sequence $\sigma(\vec{p})$, such that $\vec{p}(\vec{i}) = \vec{o}$.

3.1.1 Input Sampling

Our input sampling procedure is parameterized by $(n, \Sigma, l_{min}, l_{max})$ as defined in Section 3.1. We build the initial set of input strings $\vec{i_0} = \langle i_1^0, \ldots, i_n^0 \rangle$ ($hwwgdb, jh, \ldots wcw$ in Fig 1b) by independently sampling each input i_j^0 for $1 \leq j \leq n$. For each i_j^0, we first sample its length from a uniform discrete distribution, $|i_j^0| \sim \text{Unif}\{l_{min}, \ldots, l_{max}\}$, where $|i_j^0|$ denotes the length of the string i_j^0. We then generate the string itself by sampling $|i_j^0|$ characters independently with replacement from the alphabet Σ, i.e., $i_j^0 \sim \text{Unif}(\Sigma)^{|i_j^0|}$.

3.1.2 Program Sampling and Output Generation

Program sampling is parameterized by $(\Sigma, L_{min}, L_{max}, s_{min}, s_{max})$. We begin by programmatically selecting a cascade length L such that $L_{min} \leq L \leq L_{max}$ ($L = 4$ in Fig 1b). Next, we construct a sequence of L programs $\vec{p} = \langle p_1, \ldots, p_L \rangle$, where each program is of the form $p_k = \text{replace}(\alpha_k, \beta_k)$. Here, replace has the same semantics as Python's built-in `replace()` method for strings: the substring α_k is replaced by β_k, with the restriction that α_k is non-empty. For example in Fig 1b, the first program $c \rightarrow wa$ is parameterized by $\alpha_1 = c$ and $\beta_1 = wa$. To sample each program p_k, we generate α_k and β_k independently, following a procedure analogous to input sampling:



(a) Proposer–solver architecture

(b) Structure of a PBE data instance

Figure 1: Overview of PBEBench. (a) A symbolic proposer generates PBE instances of controllable difficulty, and a solver attempts to recover the underlying program sequence or ordering. (b) Each instance consists of $<\vec{i}, \vec{p}, \vec{o}>$ triples, with program relations encoded by the BFCC DAG and a simplified BF variant that replaces counterfactual relations with a reversed link.

1. Sample the lengths $|\alpha_k|$ and $|\beta_k|$ from a discrete uniform distribution: $|\alpha_k|, |\beta_k| \sim \text{Unif}\{s_{min}, \ldots, s_{max}\}$
2. Sample α_k uniformly from the set of substrings of length $|\alpha_k|$ in the intermediate input vector $\vec{i}_{k-1} = p_{k-1}(\ldots p_1(\vec{i}))$ (where $\text{Substr}_s(\vec{i}_{k-1})$ denotes the set of all substrings of length s in $\vec{i}_{k-1}$): $\alpha_k \sim \text{Unif}(\text{Substr}_{|\alpha_k|}(\vec{i}_{k-1}))$ (note that $\vec{i}_0 = \vec{i}$). For example in Fig 1b, for α_2, we had $|\alpha_2| = 1$, thus we would sample uniformly from substrings of length one (basically all characters) present in the intermediate inputs $\vec{i_1} = \langle hwwgdb, jh, ahwawb, xuuiib, wwaw \rangle$.
3. Sample β_k as a sequence of $|\beta_k|$ characters, each drawn independently from the uniform character distribution over the alphabet Σ (same as in Section 3.1.1): $\beta_k \sim \text{Unif}(\Sigma)^{|\beta_k|}$

The outputs are then generated by executing the program cascade $\vec{p}$ over the inputs $\vec{i}$ as $\vec{o} = \vec{p}(\vec{i})$ which as stated above is just recursive application of the replace programs $p_k(p_{k-1}(\ldots p_1(\vec{i_o})))$

3.1.3 Enforcing Complexity Constraints

We control the complexity of cascades of programs $\vec{p}$ using rejection sampling guided by the classifiers developed in Section B. Complexity is balanced along two dimensions: (1) the cascade length $L = |\vec{p}|$, and (2) the relation types present between programs in $\vec{p}$. The second dimension is encoded as a binary category string per cascade $\vec{p}$, $c_{\vec{p}} \in \{0, 1\}^4$. Each digit is a binary indicator denoting the presence or absence of feeding, bleeding, counter-feeding, and counter-bleeding relations. Instances are balanced across all 16 possible category strings, from $c_{\vec{p}} = 0000$ (no relations present, arbitrary ordering possible) to $c_{\vec{p}} = 1111$ (all relations present, ordering highly constrained).

Computing Instance Complexity. We hypothesize that the difficulty of a PBE problem $\langle \vec{i}, \vec{p}, \vec{o} \rangle$ is governed by (1) cascade length and (2) the types of relations between program pairs in $\vec{p}$ (relations formulated by phonologists and historical linguists; see Kiparsky (1968, 1971)). One of our key contributions is provably correct, automatic, symbolic classification of these relations for a given pair of programs (Appendix B). For a pair of programs (p_i,p_j) in $\vec{p}$, where p_i is applied before p_j, the possible relations are:

Feeding (F(p_i, p_j)): p_i creates substrings that enable p_j to apply.

Bleeding (B(p_i, p_j)): p_i removes substrings required by p_j.

Counter-Feeding (CF(p_i, p_j)): p_i could have fed p_j, but p_j precedes p_i.

Counter-Bleeding (CB(p_i, p_j)): p_i could have bled p_j, but p_j precedes p_i.

No Relation: p_i and p_j can be ordered arbitrarily.

Counter-relations need not be stored explicitly, since $CF(p_i, p_j)$ and $CB(p_i, p_j)$ are implied if p_j precedes p_i and $F(p_j, p_i)$ and $B(p_j, p_i)$, respectively. These relations can be visualized using a Directed Acyclic Graph (DAG), as illustrated in

Figure 1b. We show both the full DAG with all relations and a simplified DAG that indirectly encodes counter-relations via this symmetry.

Rejection Sampling to Control Complexity. To enforce balanced complexity, we use rejection sampling. Each PBE instance is first assigned a relation-type category string $c_{\vec{p}}$, and data is generated to approximate balance across the $2^4 = 16$ categories, as well as across cascade lengths between L_{min} and L_{max}. These constraints can conflict, since short cascades (e.g., $L = 2$) rarely realize higher-order relation types, making some categories unattainable. Following prior work on sound law induction (Naik et al., 2025a), we retain random sampling rather than deterministic generation or retrieval to promote diversity and avoid contamination, at the cost of high rejection rates. We introduce a patience parameter τ, denoting the number of sampling steps under both constraints; once τ ($= 100{,}000$) is exceeded, we also accept instances satisfying just one constraint. Early experiments prioritized relation-type balance, but later analysis (Appendix F.7, F.8) identified cascade length as a stronger predictor of difficulty, leading us to relax relation-type constraints after excceeding τ steps. We show the effect of varying τ on relation-type balance and generation efficiency (Appendix F.6).

3.2 Problem Solver

Prompting Strategy: Once a benchmark snapshot $\mathcal{D}$ is generated, we evaluate each LLM M by prompting it for multi-step PBE (second from bottom in Fig 1a; prompt in Appendix D.3.1) and program reordering (bottom in Fig 1a; prompt in Appendix D.3.2).

For multi-step PBE, the prompt enforces the following constraints: each program must be a Python `replace` function; both arguments α_k and β_k must be strings with $|\alpha_k|, |\beta_k| \leq s_{max}$; α_k must be non-empty; the cascade may contain at most L_{max} programs (matching the longest ground-truth cascade); and outputs must follow a strict Markdown format for reliable extraction. If a predicted cascade $\hat{\vec{p}}$ exceeds L_{max} programs, only the first L_{max} are evaluated; violations of any other constraint result in replacement with an identity program p^I, which leaves inputs unchanged. Given $\hat{\vec{p}}$, predicted outputs are computed as $\hat{\vec{o}} = \hat{\vec{p}}(\vec{i})$. Details of program extraction from LLM responses are provided in Appendix D.2.

For the program reordering task, the prompt defines BLEEDING and FEEDING relations, provides the ground-truth program cascade along with the inputs and outputs, and instructs the LLM to output only a fenced JSON block containing a permutation of indices (e.g., $[2, 1, 3]$) that recovers the original program order. This task evaluates the model's ability to reason about ordering constraints induced by BFCC relations between programs.

Scaling Solution Search. We investigate two solution-search scaling strategies, motivated by evidence of reasoning collapse and out-of-token failures in (Shojaee et al., 2025) and subsequent discussion (Lawsen, 2025): (1) sampling-budget (Li and Ellis, 2024) and (2) test-time thinking-budget scaling (Muennighoff et al., 2025). In the first, the LLM produces K candidate solutions, and any candidate consistent with the input–output examples is accepted; if none succeed, we select the candidate with the highest edit-similarity reward (Section 4.3). In the second, inspired by (Muennighoff et al., 2025), we vary a thinking budget of N tokens. However, because gpt-oss-120b does not reliably respect this constraint, even when forced to emit chain-of-thought termination tokens (Appendix E.3), we instead control the overall maximum sequence length.

4 Experiments

4.1 Dataset Creation

Using the problem proposer described in Section 3.1, we construct three synthetic benchmarks.

PBEBench-Lite: A simplified dataset with cascades of length 2 to 5 and 5 input–output pairs per instance, synthesized as $\mathcal{D}(n = 5, \Sigma = \Sigma_{\text{lite}}, L_{min} = 2, L_{max} = 5, l_{min} = 2, l_{max} = 6, s_{min} = 1, s_{max} = 3, D = 1008)$, with exactly 63 examples per relation category. The alphabet is restricted $\Sigma_{\text{lite}} = \{a, \ldots, k, u, v, w, x, y, z\}$ (lowercase letters excluding l–t).

PBEBench-Lite-Perm: A program reordering dataset with 919 instances created from PBEBench-Lite using the swapping strategy (Section 3.1).

PBEBench: a larger dataset with 50 input-output pairs per instance, synthesized as $\mathcal{D}(n = 50, \Sigma, L_{min} = 2, L_{max} = 20, l_{min} = 2, l_{max} = 6, s_{min} = 1, s_{max} = 3, D = 1216)$, with 64 instances for each cascade length. Here $\Sigma = \{a, \ldots, z, A, \ldots, Z\}$ (full alphabet in both cases).

Real-SLI: We also evaluate all open-source LLMs on real SLI data drawn from 6 proto–attested language pairs, described in more detail in Appendix C.3. To assess the predictive power of

PBEBench for SLI performance, we additionally compare it against other inductive reasoning benchmarks, including CLUTRR (Sinha et al., 2019) and SLR-Bench (Helff et al., 2025). Some other datasets generated for some ablations are described in Appendix C.2.

4.2 Models Evaluated

We evaluate a broad range of state-of-the-art LLMs spanning multiple model families and architectures. We cover models from leading developers including OpenAI (GPT, o-series, gpt-oss), Anthropic (Claude series), Google (Gemini), Qwen (Qwen2.5, Qwen3, QwQ), DeepSeek (R1-Distill-Qwen), Mistral (Codestral). The models differ in scale, reasoning specialization (thinking vs. non-thinking), architectural choices (dense vs. MoE), and source availability (open vs. closed). A full list of models with their attributes is provided in Table 4, covering the breadth of model families and capabilities.

4.3 Evaluation Metrics

For the multi-step PBE task, since a given input vector $\vec{\imath}$ may be mapped to the output vector $\vec{o}$ by multiple program cascades $\vec{p}$, we evaluate model predictions using metrics based on functional equivalence, following prior work in programming by example (Li and Ellis, 2024) and historical linguistics (Hoenigswald, 1960). We execute the model-generated cascade $\hat{\vec{p}}$ on the inputs and compare the predicted outputs $\hat{\vec{o}} = \hat{\vec{p}}(\vec{\imath})$ with $\vec{o}$.

We use two main metrics: a coarse-grained solve rate (`Pass@1`) and a fine-grained normalized edit similarity (`Edit_Sim`):

$$\text{pass@1} = \frac{1}{|\mathcal{D}|} \sum_{\vec{o},\vec{\imath} \in \mathcal{D}} 1_{\hat{\vec{p}}(\vec{\imath})=\vec{o}}$$

$$\text{Edit\_Sim} = \frac{1}{|\mathcal{D}|} \sum_{\vec{o},\vec{\imath} \in \mathcal{D}} 1 - \frac{\text{dist}(\hat{\vec{p}}(\vec{\imath}), \vec{o})}{\text{dist}(\vec{\imath}, \vec{o})},$$

where dist denotes the summed Levenshtein edit distance over corresponding strings. We additionally report the `Valid_Rate`, the proportion of generated programs that satisfy all prompt constraints and `Complexity` that sums up the length of the substrings α_k and β_k in the model predicted program cascade and averages it across PBE instances. For the program reordering task, we evaluate accuracy (Acc) by applying the predicted permutation $\hat{\sigma}$ to the permuted program sequence and checking whether $\hat{\sigma}(\sigma(\vec{p}))(\vec{\imath}) = \vec{o}$ (Section C.4). We also report unique accuracy on instances with a single valid solution.

Model	Acc	UAcc
Codestral-22B	34.3	51.2
Qwen2.5-32B-Instruct	31	39.3
Qwen2.5-Coder-32B-Instruct	34.7	49.6
Qwen3-32B	36	43
Qwen3-Coder-30B-A3B-Instruct ■	44.1	63.2
DeepSeek-R1-Distill-Qwen-32B ★	75.5	92.1
Qwen3-30B-A3B ★■	68	91.3
QwQ-32B ★	74.9	94.6
Qwen3-32B (with CoT) ★	77.9	93.8
gpt-oss-20b ★■	86.3	92.6
gpt-oss-120b ★■	97.5	98.8
Claude-4-Sonnet ✷	80.2	91.3
Claude-4.5-Sonnet ✷	85.1	92.1
Claude-4-Sonnet (Thinking) ★✷	91.6	97.5
Claude-4.5-Sonnet (Thinking) ★✷	97.5	99.2
o3-mini ★✷	63.4	83.7
o4-mini ★✷	80.2	97.5
Gemini 2.5 Flash ★✷	92.8	89.7
GPT-5 ★✷	99.7	99.6

Table 1: **PBEBench-Lite Program Reordering:** We compute accuracy (`Acc`) based on functional correctness of the unscrambled program sequence on the full data and the unique solution subset (`UAcc`). ■ indicates a mixture-of-experts (or MoE) model. ★ indicates a reasoning model. ✷ indicates unknown architecture.

5 Results

5.1 PBEBench-Lite Performance

Table 2 reports `Pass@1`, `Edit_Sim`, `Valid_Rate` (expressed as percentages), and `Complexity` for all models (Section 4.2). Hyperparameters used in model evaluations are listed in Table 8. Claude-4 Opus (Thinking) was evaluated on only 20% of the data due to high API costs (run costs are reported in Appendix D.6). Reasoning models, both open and closed-source, consistently outperform non-reasoning models, with a larger gap among open-source models. The top-performing models are gpt-oss-120b (open) and GPT-5 (closed). Surprisingly, Qwen3-30B-A3B and o3-mini produce the simplest programs yet perform poorly, suggesting underthinking, while Claude-4-Sonnet and Qwen3-32B without CoT produce the most complex programs, indicative of overthinking. The ground-truth programs have a mean complexity

Model	Pass@1	Edit Sim	Complexity	Valid Rate
Codestral-22B	1.1	-1.1	15.4	82.5
Qwen2.5-32B-Instruct	3	12.5	12.5	82.9
Qwen2.5-Coder-32B-Instruct	4.1	18.8	12.67	68.9
Qwen3-32B	1.8	9.6	14.89	76.5
Qwen3-Coder-30B-A3B-Instruct ■	3.8	8.6	3.25	81.3
DeepSeek-R1-Distill-Qwen-32B ★	22.4	34.9	7.43	87.1
Qwen3-30B-A3B ★■	28.9	33.6	**4**	**99.1**
QwQ-32B ★	36	40.9	4.72	94.9
Qwen3-32B (with CoT) ★	41.9	50.3	6.57	96.8
gpt-oss-20b ★■	40.6	46.2	6.96	99
gpt-oss-120b ★■	**62.5**	**69.9**	10.93	92.5
Claude-3.5-Sonnet ✷	18.5	44.3	11.87	82.1
Claude-3.7-Sonnet ✷	23.2	50	12.05	84.1
Claude-4-Sonnet ✷	29.7	58.7	13.35	77.2
Claude-3.7-Sonnet (Thinking) ★✷	36.6	61.5	12.94	0.819
Claude-4-Sonnet (Thinking) ★✷	35.8	60.8	13.71	78.2
Claude-4-Opus (Thinking) ★✷	53.9	75.2	13.44	85.6
o3-mini ★✷	19.6	19.8	**1.65**	**99.5**
o4-mini ★✷	36.3	37.4	4.08	97.3
Gemini 2.5 Flash ★✷	58.6	65.6	8.36	79
GPT-5 ★✷	**72.4**	**76.5**	10.58	92.9

Table 2: **PBEBench-Lite Performance:** We compute the `Pass@1` and `Edit_Sim` as the coarse and fine-grained evaluation, respectively, for each model. ■ indicates a mixture-of-experts (or MoE) model. ★ indicates a reasoning model. ✷ indicates unknown architecture. * indicates evaluated on 20% of the dataset due to cost.

of 11.63, indicating that even the best-performing models, GPT-5 and gpt-oss-120b, generate simpler programs than the ground truth. Factorial analysis of QwQ-32B and GPT-5 identifies cascade length as the strongest negative predictor of `Pass@1`. We further find that feeding and bleeding decrease `Pass@1`, whereas counter-feeding and counter-bleeding have no significant effect (Appendix F.7). Finally, confusion-matrix analyses of predicted versus ground-truth cascade lengths and relation types (Appendix F.10, F.11) show that models favor simpler cascades with fewer relations and programs, typically succeeding only when a functionally equivalent solution exists.

5.2 PBEBench-Lite-Perm Performance

Table 1 reports `Acc` and `UAcc` for the PBEBench-Lite-Perm program reordering dataset. We could not run Claude-3.5-Sonnet and Claude-3.7-Sonnet due to deprecation, and Claude-4-Opus due to budget constraints, and therefore evaluate Claude-4.5-Sonnet with and without reasoning. While LLMs may sometimes bypass BFCC relations, this task requires explicit reasoning to recover correct ordering, with the unique-solution subset admitting only one valid order. Using PBEBench-Lite with short cascades (2–5) allows some models to enumerate all orderings, simplifying the task relative to full multi-step PBE. GPT-5 nearly solves the task with 99.7% accuracy, whereas Qwen2.5-32B-Instruct performs worst at 31%. Logistic regression analysis (Table 17) reveals that LLMs predictably struggle more with longer cascades and counter relations, like COUNTER-FEEDING. Finally, `UAcc` is generally higher than `Acc`, explained by the fact that 182 of 242 unique-solution instances have cascade length 2, where random guessing already yields 50% accuracy. We also present a qualitative analysis of the various solution strategies adopted by the evaluated models in Appendix F.9.

5.3 PBEBench Performance

We evaluate the strongest open and closed-source models (gpt-oss-120b and GPT-5) on PBEBench across cascade lengths ranging from 2 to 30. GPT-5 consistently outperforms gpt-oss-120b, maintain-

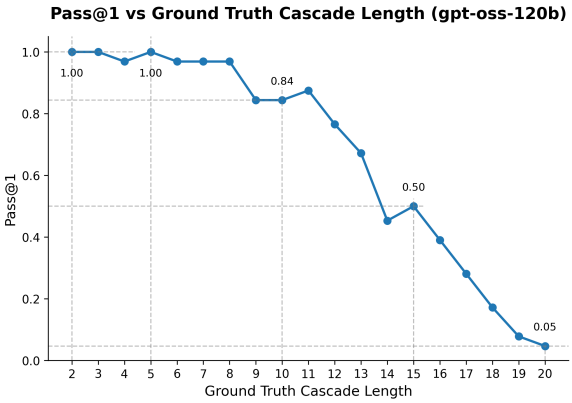
ing non-trivial Pass@1 at longer cascades where gpt-oss-120b performance collapses. At length 20, GPT-5 achieves moderate accuracy with a small sampling budget, while gpt-oss-120b drops to near-zero performance even with larger budgets. Performance declines monotonically with cascade length for both models, revealing clear reasoning limits. Trends across lengths are summarized in Fig. 2a and Fig. 2b. A regression analysis further confirms cascade length as a dominant negative predictor of success, with the presence of specific BFCC relations associated with failure or success. Full results and analyses are deferred to Appendix F.1.

5.4 Ablations and Scaling

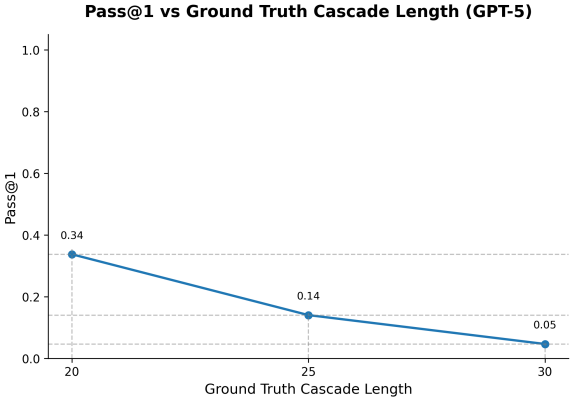
We perform ablations using PBEBench-Lite and its variant PBEBench-Lite-MoreEg (Appendix C.2) to analyze the effects of just increasing input-output examples on the gpt-oss models. Increasing the number of examples per PBE instance results in a slight reduction in performance. We also analyze the effect of the scaling strategies: sampling budget and thinking budget on GPT-5 and gpt-oss-120b on the harder PBEBench data. We observe consistent scaling behavior for both models, with rapid initial gains from increased compute, followed by diminishing returns and "hitting a wall" for longer ground truth cascades. Detailed quantitative results, scaling curves, and variance analyses are provided in Appendix F.2.

5.5 Real SLI Performance

We evaluate all open-source models on real SLI data (Appendix C.3), with results in Table 13. We additionally report performance on CLUTRR (Sinha et al., 2019) and SLR-Bench (Helff et al., 2025) in Appendix F.3. Motivated by our hypothesis that SLI is bottlenecked by multi-step inductive reasoning (Section 1), we measure Pearson rank correlations between model rankings on real SLI and those induced by PBEBench-Lite, CLUTRR, and SLR-Bench (Table 14). PBEBench-Lite shows the strongest correlation with real SLI ($r = 0.8273$, $p = 0.001677$), followed closely by SLR-Bench ($r = 0.8091$, $p = 0.002559$). However, SLR-Bench is sensitive to Prolog coding proficiency, with Codestral-22B achieving high syntax scores but weak overall performance relative to Qwen models, whose reasoning is stronger despite poorer Prolog syntax (Table 11). By relying on Python, PBEBench reduces syntax bottlenecks that would otherwise underestimate inductive capability.



(a) gpt-oss-120b (sampling budget: 32, max sequence length: 16384, cascade: 2-20)



(b) GPT-5 (sampling budget: 4, max completion tokens: 65536, reasoning (medium), cascade: 20, 25, 30)

Figure 2: **Performance across Cascade Lengths on PBEBench:** Pass@1 for gpt-oss-120b and GPT-5 for various ground truth cascade lengths, a key difficulty measure, shows where inductive reasoning fails and problems become nearly unsolvable despite high compute budgets.

6 Discussion

After analyzing the results, we address all the hypotheses in Section 1. **H1: LLMs struggle with multi-step inductive reasoning.** As discussed in Section 5.1, we see non-reasoning LLMs struggle to solve even simple multi-step PBE problems, and most open-source reasoning models outperform closed-source non-reasoning ones. Additionally, even the best performing reasoning models, GPT-5 and gpt-oss-120b, break when evaluated on the hard PBEBench data (Section 5.3) despite inference-time scaling strategies (Section 5.4). **H2: Difficulty in our dataset is driven by interaction structure.**: We find that increasing per-step difficulty by adding more examples has little effect on performance (Section 5.4). In contrast, factorial and logistic regression analyses (Sections 5.1,5.3)

and performance degradation on longer cascades in PBEBench (Figs. 2a,2b) show that LLMs struggle primarily with long cascades and complex BFCC interactions. Moreover, many models, including reasoning LLMs, fail to recover the correct program order in PBEBench-Lite-Perm even when given the correct shuffled programs for cascades of length 2 to 5, indicating that task difficulty arises from interactions within the cascade rather than individual steps. **H3: Performance on PBEBench predicts performance on real SLI**: As discussed in Section 5.5, even the simpler PBEBench-Lite has higher correlation with the model rankings on the real SLI data compared to CLUTRR and SLR-Bench, indicating the predictive power of our approach. **H4: Scaling the thinking budget is more effective than increasing the sampling budget.** A direct comparison for gpt-oss-120b shows that increasing sequence length with sufficient sampling is more time and cost-efficient than increasing sampling budget alone, as it reduces wasted attempts from unterminated chains of thought.

Computational complexity and symbolic solvers. A key question is whether PBEBench instances are inherently hard or simply poorly matched to current models. Under a naive enumeration view, the search space grows exponentially with cascade length. Each rewrite rule selects substrings over an alphabet Σ (e.g., of length 1–3), yielding $O(|\Sigma|^k)$ possibilities per side and $O(|\Sigma|^{2k})$ candidate rules. A cascade of length L therefore induces a hypothesis space of size $O(|\Sigma|^{2kL})$, which is exponential in L. While this is not a formal hardness proof, it highlights that exhaustive search quickly becomes intractable as cascade length increases. Unlike classical program synthesis settings such as FlashFill, our benchmark does not assume invertibility or structural constraints that enable dynamic programming or polynomial-time synthesis. Rule interactions are order-sensitive and do not decompose into independent subproblems, limiting the applicability of DP formulations. Constraint-based or symbolic solvers could, in principle, enumerate and verify candidate cascades, but absent a strong pruning structure, they still operate over an exponential search space. This positions PBEBench in a regime where both neural and symbolic methods require non-trivial search strategies.

LLMs as heuristic search and solution multiplicity. Given the combinatorial nature of the task, we do not view LLMs as complete solvers, but rather as heuristic proposal mechanisms that impose structure on the search space, motivating hybrid neuro-symbolic approaches. We also note that valid cascades are not necessarily unique, as multiple programs may satisfy the same input-output examples. Importantly, the source of multiplicity differs across settings. In the full PBEBench task, both the set of programs and their ordering are unknown, leading to a large (and potentially unbounded) space of valid solutions. In contrast, PBEBench-Lite-Perm isolates *ordering* by providing the correct set of programs and requiring only recovery of a valid permutation under BFCC constraints. In this restricted setting, if a cascade has length n and k rules participate in BFCC constraints, the number of valid orderings is upper bounded by $(n-k+1)!$, since constrained rules must preserve relative order while the remaining blocks can be permuted. For PBEBench-Lite-Perm ($n \leq 5$ with at least one BFCC constraint), this yields at most $4! = 24$ solutions, which is modest and could be enumerated. Despite this, performance still degrades with cascade length and counter-relations, indicating that interaction structure, rather than solution multiplicity, is the primary source of difficulty.

7 Conclusion and Future Work

We show that the proposed benchmark is challenging for many models, especially non-reasoning LLMs, because despite simple individual PBE steps, its sequential structure and BFCC-induced ordering constraints create emergent complexity that requires long chains of thought at test time. Increasing cascade length or interaction complexity breaks even the strongest reasoning models, including GPT-5 and gpt-oss-120b. It is also practically meaningful: PBEBench-Lite performance best predicts real SLI performance, suggesting that LLMs' difficulty with multi-step inductive reasoning partly explains their poor real-world SLI results. In future work, we plan to synthesize progressively harder training data using PBEBench and explore hybrid neuro-symbolic data generation.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. 2504019. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily

reflect the views of the National Science Foundation.

Limitations

Despite the contributions of this work, several limitations remain:

1. We primarily evaluate the strongest open and closed-source models on PBEBench. Due to time and compute constraints, we don't extensively benchmark weaker models; however, results on PBEBench-Lite indicate that such models consistently underperform relative to gpt-oss-120b and GPT-5, especially on longer cascades.
2. Owing to budget and time constraints, evaluations on CLUTRR, SLR-Bench, and real SLI data are limited to open-source models.
3. For PBEBench-Lite, PBEBench-Lite-MoreEg, CLUTRR, and SLR-Bench, we use a single sample per problem (sampling budget of 1), which may underestimate performance under larger sampling budgets.
4. For real SLI, we simplify the task by chunking cases with more than 200 examples into subsets of 50, relaxing program constraints by allowing longer cascades and larger substring rewrites. While this could enable degenerate solutions, most models still perform poorly even with increased thinking and sampling budgets.
5. Due to cost, GPT-5 is evaluated only on the most challenging cascade lengths and with limited sampling budgets, restricting comprehensive analysis across settings.
6. The random sampling procedure cannot perfectly balance constraints such as relation types and cascade lengths, particularly for long cascades where valid examples are rare, leading to high rejection rates and exhaustion of the patience parameter τ. Future work could explore more efficient sampling strategies.
7. In PBEBench, individual programs modify relatively few examples on average, which is less representative of real sound law induction, where rules often apply to many words simultaneously.

Ethics Statement

Our work introduces a provably correct, fully automated, and scalable pipeline for generating multi-step Programming by Example (PBE) problems, enabling the evaluation of inductive reasoning in Large Language Models without human supervision. As no human subjects are involved (apart from the individuals who contributed the published data for the SLI task), we do not anticipate ethical concerns. Moreover, since the dataset focuses solely on assessing abstract inductive reasoning in LLMs, it is unlikely to be misused for harmful purposes.

References

Mistral AI. 2024. Codestral-22b-v0.1. https://huggingface.co/mistralai/Codestral-22B-v0.1. Released under the Mistral AI Non-Production License (MNPL); retrieved May 9, 2025.

Rajeev Alur, Dana Fisman, and Rishabh Singh. 2013. Syntax-guided synthesis. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8.

Anthropic. 2024. Sonnet. https://console.anthropic.com/. Output generated by Claude 3.5; retrieved May 9, 2025.

Anthropic. 2025a. Claude opus 4 model overview. Anthropic model card / announcement. Released May 22, 2025; 200,000 token context window.

Anthropic. 2025b. Claude sonnet 4 model overview. Anthropic Models Overview Webpage. API name: `claude-sonnet-4-20250514`, 200K token context window, high-performance model with exceptional reasoning and efficiency.

Anthropic. 2025. Sonnet. https://console.anthropic.com/. Output generated by Claude 3.7; retrieved May 9, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, and et al. 2021. Program synthesis with large language models. In *International Conference on Learning Representations (ICLR)*. ArXiv:2108.07732.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR)*.

Mark Chen, Jerry Tworek, Heewoo Jun, and et al. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Karl Cobbe, Vineet Kosaraju, Timothy Lillicrap, Chris Paine, Jacob Hilton, and et al. 2021. Training verifiers to solve math word problems. In *Advances in Neural Information Processing Systems*. ArXiv:2107.00250.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. *Preprint*, arXiv:1703.07469.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. 2019. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *North American Chapter of the Association for Computational Linguistics*.

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. Refactorbench: Evaluating stateful reasoning in language agents through code. In *The Thirteenth International Conference on Learning Representations*.

Daniel Gildea and Daniel Jurafsky. 1995. Automatic induction of finite state transducers for simple phonological rules. In *33rd Annual Meeting of the Association for Computational Linguistics*, pages 9–15, Cambridge, Massachusetts, USA. Association for Computational Linguistics.

Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, page 13–24, New York, NY, USA. Association for Computing Machinery.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 317–330. ACM.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Lukas Helff, Ahmad Omar, Felix Friedrich, Antonia Wüst, Hikaru Shindo, Tim Woydt, Rupert Mitchell, Patrick Schramowski, Wolfgang Stammer, and Kristian Kersting. 2025. Slr: Automated synthesis for scalable logical reasoning. In *First Workshop on Foundations of Reasoning in Language Models*.

Henry M Hoenigswald. 1960. Language change and linguistic reconstruction. *(No Title)*.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. 2025. Testgeneval: A real world unit test generation and test completion benchmark. *Preprint*, arXiv:2410.00752.

Daniel Keysers, Nathanael Schärli, Nathan Scales, and et al. 2020. Measuring compositional generalization: A comprehensive method on realistic data. *Transactions of the Association for Computational Linguistics*, 8:328–344.

Paul Kiparsky. 1968. Linguistic universals and linguistic change. In Emmon W. Bach and Robert Thomas Harms, editors, *Universals in Linguistic Theory. (Edited by Emmon Bach, Robert T. Harms ... Contributing Authors, Charles J. Fillmore ... Paul Kiparsky ... James D. McCawley.)*., pages 170–202. Holt, Rinehart, and Winston.

Paul Kiparsky. 1971. Historical linguistics. In William Orr Dingwall, editor, *A Survey of Linguistic Science*. ERIC.

Brenden M. Lake and Marco Baroni. 2018. Generalization without systematicity: Compositional generalization in recurrent networks. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 2879–2888.

A. Lawsen. 2025. Comment on the illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *Preprint*, arXiv:2506.09250.

Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. *Preprint*, arXiv:2404.13340.

Wen-Ding Li and Kevin Ellis. 2024. Is programming by example solved by LLMs? In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Kaijing Ma, Xinrun Du, Yunran Wang, Haoran Zhang, Zhoufutu Wen, Xingwei Qu, Jian Yang, Jiaheng Liu, Minghao Liu, Xiang Yue, and 1 others. 2024. Kor-bench: Benchmarking language models on knowledge- orthogonal reasoning tasks. *arXiv preprint arXiv:2410.06526*.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Atharva Naik, Darsh Agrawal, Hong Sng, Clayton Marr, Kexun Zhang, Nathaniel R Robinson, Kalvin Chang, Rebecca Byrnes, Aravind Mysore, Carolyn Rose, and David R Mortensen. 2025a. Programming by examples meets historical linguistics: A large language model based approach to sound law induction. *Preprint*, arXiv:2501.16524.

Atharva Naik, Darsh Agrawal, Hong Sng, Clayton Marr, Kexun Zhang, Nathaniel R Robinson, Kalvin Chang, Rebecca Byrnes, Aravind Mysore, Carolyn Rose, and 1 others. 2025b. Programming by examples meets historical linguistics: A large language model based approach to sound law induction. *arXiv preprint arXiv:2501.16524*.

Atharva Naik, Kexun Zhang, Nathaniel Robinson, Aravind Mysore, Clayton Marr, Hong Sng, Rebecca Byrnes, Anna Cai, Kalvin Chang, and David Mortensen. 2024. Can large language models code like a linguist?: A case study in low resource sound law induction. *arXiv preprint arXiv:2406.12725*.

OpenAI. 2024. Gpt o3-mini. `https://platform.openai.com/`. Output generated by GPT o3-mini; retrieved May 9, 2025.

OpenAI. 2025. Gpt-5. Official model release; OpenAI blog post.

OpenAI. 2025. Gpt o4-mini. `https://platform.openai.com/`. Output generated by GPT o4-mini; retrieved May 9, 2025.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. arXiv preprint, model card. *Preprint*, arXiv:2508.10925.

Yunfan Shao, Linyang Li, Yichuan Ma, Peiji Li, Demin Song, Qinyuan Cheng, Shimin Li, Xiaonan Li, Pengyu Wang, Qipeng Guo, Hang Yan, Xipeng Qiu, Xuanjing Huang, and Dahua Lin. 2025. Case2Code: Scalable synthetic data for code generation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 11056–11069, Abu Dhabi, UAE. Association for Computational Linguistics.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. 2025. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *Preprint*, arXiv:2506.06941.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L Hamilton. 2019. Clutrr: A diagnostic benchmark for inductive reasoning from text. *arXiv preprint arXiv:1908.06177*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, , and Jason Wei. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Qwen Team. 2025a. Qwen3.

Qwen Team. 2025b. Qwq-32b: Embracing the power of reinforcement learning.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart Van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Zhe Zhang, Runlin Liu, Aishan Liu, Xingyu Liu, Xiang Gao, and Hailong Sun. 2025. Dynamic benchmark construction for evaluating large language models on real-world codes. *Preprint*, arXiv:2508.07180.

Appendix

The Appendix is organized as follows:

1. Appendix A discusses steps taken to ensure reproducibility.
2. Appendix B provides the theory and proofs for the relation type classifiers used to control the distribution.
3. Appendix C details the benchmark and metrics, including licensing, statistics, and related information.
4. Appendix D gives additional methodological details, including prompts, model selection, licensing, run costs, algorithm pseudocode, snapshots for closed-source models, and the K-fold evaluation used to simulate GPT-5's sampling budget.
5. Appendix E outlines experimental details such as computational environment, inference parameters, and strategies for CoT truncation/control with gpt-oss-120b.
6. Appendix F presents additional results, including data generation efficiency, factorial and logistic regression analyses of instance difficulty, confusion matrices comparing dataset ground truth and model predictions, and extended tables for scaling experiments, effects of additional examples, and performance breakdowns by cascade length.

A Reproducibility Statement

To facilitate the reproduction of our results, we thoroughly document all components of our work. The BFCC relation types, their detectors (Algorithm 2), and proofs of correctness are presented in Appendix B. The rejection-sampling-based data generation process is both described (Section 3.1) and formally specified (Algorithm 1). Additionally, the algorithm for the feeding-bleeding swap permutation for creating the program reordering data is documented in Algorithm 3. Detailed statistics and parameters for all generated datasets appear in Section 4.1 and Appendix C.2. Our prompting and program extraction strategy, including scaling techniques, is reported in Section 3.2, while the prompt template is provided in Appendix D.3, and inference parameters for each model are given in Appendix E.2. For closed-source models, we use specific dated snapshots/checkpoints of models as recorded in Appendix D.7 to aid with reproducibility. To estimate the amount of variance between runs/experiments, we do a small controlled experiment for gpt-oss-120b (Table 26), and compute k-fold average for GPT-5 (Appendix D.8) The evaluation procedure and metrics are described in Section 4.3 and Section 3.2.

B Theoretical Framework

This section provides the proof of correctness of our proposed method for automatically classifying the type of relation between any pair of string-rewrite programs, which is one of our novel contributions.

We propose the function $\mathsf{feeds}(\cdot,\cdot)$ (equation 1), which classifies pairs of rules as feeding or not feeding.

where $\mathsf{Pref}(s)$, $\mathsf{Suff}(s)$, and $\mathsf{Substr}(s)$ are the multisets of prefixes, suffixes, and substrings of s, respectively.

Definition B.1 (Feeding)**.** Feeding is a relation between pairs of rules $p_i = s_i \rightarrow t_i$ and $p_j = s_j \rightarrow t_j$, such that $\exists s, t \in \Sigma^*$ such that $s \xrightarrow{p_i} t$ and t includes a string w that meets the structural description of p_j but is not present in s.

Definition B.2 (Bleeding)**.** Bleeding is a relation between pairs of rules $p_i = s_i \rightarrow t_i$ and $p_j = s_j \rightarrow t_j$, such that $\exists s, t' \in \Sigma^*$ such that $s_i \xrightarrow{p_i} t_i$ and s_i includes a string w that meets the structural description of p_j but is not present in t_i.

Definition B.3 (Substr)**.** $\mathsf{Substr}(s)$ denotes the multiset of substrings of s, counting multiple occurances separately.

Lemma 1: If $\mathsf{feeds}(p_i, p_j)$ then p_i feeds p_j.

Proof. Given $u, v, o, s_i, t_i, s_j, t_j \in \Sigma^*$, $s_i \xrightarrow{p_i} t_i$, and $s_j \xrightarrow{p_j} t_j$ there are four types of transformations of u by applying p_i that will yield v such that $s_j \sqsubseteq v$ (where $\sqsubseteq$ indicates "is a substring of"). (1) **Deletion.** Assume that $t_i = \varepsilon$. $\exists wx \in \Sigma^+$ such that $ws_ix \xrightarrow{p_i} xw$. If $s_j = xw$ then p_i feeds p_j. (2) **Containment.** $t_i \sqsubseteq s_j \wedge t_i \not\sqsubseteq s_i$, $\exists w, x \in \Sigma+$ such that $w \xrightarrow{p_i} x \wedge s_j \sqsubseteq x \wedge s_j \not\sqsubseteq x$. (3) **Subsumption.** Assume that $s_j \in \mathsf{Substr}(t_i) \setminus \mathsf{Substr}(s_i)$. Given $s_i \xrightarrow{p_i} t_i$, t_i will always contain instances of s_j not present in s_i, entailing that p_i feeds p_j. (4) **Completion.** Assume that $t_i = uo$ and $s_j = ov$ (so that o is a suffix of t_i and a prefix of s_j). $s_iov \xrightarrow{p_i} t_iov = uov = us_j$, entailing that p_i feeds p_j (as with $t_i = ou$ and $s_j = vo$, *mutatis mutandis*). □

Lemma 2: If $\neg\mathsf{feeds}(p_i, p_j)$ then p_i does not feed p_j

Proof. Given $s_i, t_i, s_j, t_ju \in \Sigma^*$, assume for the sake of contradiction two rewrite rules $s_i \xrightarrow{p_i} t_i$ and $s_j \xrightarrow{t}_j$ such that p_i feeds p_j but s_i, t_i, and s_j do not satisfy any of the following conditions: **Deletion.** $s_i \neq \varepsilon \vee s_j \neq wx \forall w, x \in \Sigma^+$, **Containment.** $t\neg \sqsubseteq s_j \vee t_i \sqsubseteq s_i$ **Subsumption.** s_j does not occur in t_i except where it occurs in s_i. **Completion.** $\nexists u, o, v$ such that$(t_i = ou \wedge s_j = vo) \vee t_i = uo \wedge s_j = ov$. Either t_i is a non-empty string neither containing nor being contained by s_j and sharing no prefix or suffix with s_j or replacing s_i with t_i derives no instances of s_j. The first case must be false, since the conditions exhaust the transformations that could yield a string containing s_j. The second case must be false, because it contradicts the definition of feeding. □

Theorem 3 (Feeding)**:** A rule $s_i \rightarrow t_i$ feeds a rule $s_j \rightarrow t_j$ iff $\mathsf{feeds}(s_i \rightarrow t_i, s_j \rightarrow t_j)$

Proof. Given two rules $p_i = s_i \rightarrow t_i$ and $s_j \rightarrow t_j$, Lemma 1 proves by enumerating cases that each of the conditions defined for $\mathsf{feed}(p_i, p_j)$ are sufficient for establishing that p_i feeds p_j. Lemma 2 proves by enumerating cases than p_i does not feed p_j if none of these conditions are satisfied. □

$$\mathsf{feeds}(s_i \to t_i, s_j \to t_j) = \begin{cases} \top & t_i = \varepsilon \wedge |s_j| > 1 \\ \top & t_i \in \mathsf{Substr}(s_j) \wedge t_i \notin \mathsf{Substr}(s_i) \\ \top & t_j \in (\mathsf{Substr}(t_i) \setminus \mathsf{Substr}(s_i)) \\ \top & \mathsf{Pref}(t_i) \setminus \mathsf{Substr}(s_i) \cap \mathsf{Suff}(s_j) \neq \emptyset \\ \top & \mathsf{Suff}(t_i) \setminus \mathsf{Substr}(s_i) \cap \mathsf{Pref}(s_j) \neq \emptyset \\ \bot & \text{otherwise} \end{cases} \tag{1}$$

Theorem 4 (Bleeding)**:** A rule $p_i = s_i \to t_i$ bleeds a rule $p_j = s_j \to t_j$ iff $\mathsf{feeds}(t_i \to s_i, s_j \to t_j)$

Proof. if $\exists u, v \in \Sigma^*, u \xrightarrow{t_i \to s_i} v$ such that $s_j \sqsubseteq v \wedge s_j \not\sqsubseteq u$, it follows that mapping $s_i \xrightarrow{p_i} t_i$ bleeds p_j (where $s_j \xrightarrow{p_j} t_j$). □

C Benchmark Details

In this section, we discuss issues such as licensing and the distributional statistics of all the data snapshots created and used in our work.

C.1 Licensing

We create a provably correct, fully automated, and scalable pipeline for generating multi-step Programming by Example (PBE) problems, enabling the evaluation of inductive reasoning in LLMs. We plan to release all the benchmark snapshots (PBEBench-Lite, PBEBench-Lite-MoreEg, PBEBench, PBEBench (25, 30)) under the CC BY-SA 4.0 license. Additionally, we produce code that allows you to generate more snapshots, which we also release under the MIT license.

C.2 Benchmark Statistics

We report the distributional statistics (like distribution of ground truth cascade lengths or relation types) of all the benchmark snapshots used in this paper below:

PBEBench-Lite: We generate a relation type balanced dataset with 1008 instances, 5 examples per PBE problem, and 63 instances per relation type category with cascade lengths ranging from 2 to 5. The alphabet spans $\Sigma_{\text{lite}} = \{a, \ldots, k, u, v, w, x, y, z\}$, each input example contains 2 to 6 letters, and each rule has 1-3 characters on either side of the replace function. The distribution of cascades and relation types is shown in Fig. 3 and Fig. 4.

PBEBench-Lite-MoreEg: We generate a relation type balanced dataset with 240 instances, 50 examples per PBE problem and 15 instances per relation type category with cascade lengths ranging from 1 to 5. The alphabet spans $\Sigma_{\text{lite}} = \{a, \ldots, k, u, v, w, x, y, z\}$, each input example contains 2 to 6 letters and each rule has 1-3 characters on either side of the replace function. The distribution of cascades and relation types is shown in Fig. 5 and Fig. 6.

PBEBench: We generate a cascade balanced dataset with 1216 instances, 50 examples per PBE problem, and 64 instances per cascade length ranging from 2 to 20. The alphabet spans $\Sigma = \{a, \ldots, z, A, \ldots, Z\}$, each input example contains 2 to 6 letters and each rule has 1-3 characters on either side of the replace function. The distribution of cascades and relation types is shown in Fig. 7 and Fig. 8.

PBEBench (25, 30): We generate a cascade balanced dataset with 128 instances, 50 examples per PBE problem, and 64 instances per cascade for cascade length of 25 and 30. The alphabet spans $\Sigma = \{a, \ldots, z, A, \ldots, Z\}$, each input example contains 2 to 6 letters and each rule has 1-3 characters on either side of the replace function. The distribution of cascades and relation types is shown in Fig. 9 and Fig. 10.

C.3 Real SLI Benchmark

We construct a benchmark of 21 instances derived from real SLI data, covering 6 proto-language and attested-language pairs: prototangkhulic-huishu, austronesian-hawaiian, austronesian-niue, austronesian-tongan, austronesian-rarotongan, and austronesian-samoan. Our data is derived from (Naik et al., 2025a, 2024) original datasets. Each instance consists of 50 example input and output pairs and uses the same prompt template as the multi-step PBE task prompt employed for all synthetic datasets described above. However, the

prompt constraints permit program cascades of up to 50 programs, the sizes of α_k and β_k can be up to 5, and each LLM is allotted 32 solution attempts (sampling budget) per PBE instance to better reflect the real-world complexity of low-resource SLI. We do note a limitation of these assumptions, as for some language pairs, such as prototangkhulic-huishu, certain input words are shorter than 5 characters, and in principle an LLM could exploit this by constructing a single string rewrite rule that replaces the entire input word with the target output word. Nevertheless, despite this potential for gaming, the results indicate that most LLMs struggle with the task, and even the best-performing gpt-oss-120B model achieves a Pass@1 of only 0.33, despite a sampling budget of 32.

C.4 Metrics Details

For the multi-step PBE task, since a given list of inputs ($\vec{\imath}$) could be transformed into the outputs ($\vec{o}$) by multiple program cascades $\vec{p}$, we utilize metrics based on functional equivalence. This is in line with programming by example literature (Li and Ellis, 2024) and historical linguistics (Hoenigswald, 1960).

Concretely, we execute the model-generated solution $\hat{\vec{p}}$ on the inputs, treating the input-output pairs as test cases. We then compare the predicted outputs $\hat{\vec{o}} = \hat{\vec{p}}(\vec{\imath})$ with the ground-truth outputs $\vec{o}$ at two different levels of granularity: (1) coarse-grained evaluation, corresponding to solve rate (`Pass@1`), and (2) fine-grained evaluation, corresponding to normalized edit similarity (`Edit_Sim`).

Both metrics perform element-wise comparisons on the strings in the output vectors:

Coarse-grained Metric (`Pass@1`): This metric corresponds to `Pass@1` as introduced in (Chen et al., 2021b).

$$\text{pass@1} = \frac{1}{|\mathcal{D}|} \sum_{\vec{o},\vec{\imath} \in \mathcal{D}} 1_{\hat{\vec{p}}(\vec{\imath})=\vec{o}}$$

Here $1_{\hat{\vec{p}}(\vec{\imath})=\vec{o}}$ is an indicator variable which equals 1 when $\hat{\vec{p}}(\vec{\imath}) = \vec{o}$ and 0 otherwise.

Fine-grained Metric (`Edit_Sim`): This metric is identical to the Reward@1 metric used by (Naik et al., 2025b).

$$\text{Edit\_Sim} = \frac{1}{|\mathcal{D}|} \sum_{\vec{o},\vec{\imath} \in \mathcal{D}} 1 - \frac{\text{dist}(\hat{\vec{p}}(\vec{\imath}), \vec{o})}{\text{dist}(\vec{\imath}, \vec{o})}$$

Here dist denotes the total Levenshtein edit distance, summed across the corresponding strings in the predicted and ground-truth output vectors.

In addition to these performance metrics, we evaluate the proportion of programs generated by an LLM that are valid, meaning they satisfy all constraints specified in the prompt (see Section 3.2). We refer to this metric as the `Valid_Rate`.

Finally, for the program reordering task, given a model-predicted permutation $\hat{\sigma}$, we apply it to the originally permuted program sequence $\sigma(\vec{p})$ and evaluate whether it correctly recovers the original outputs. Specifically, we execute $\hat{\sigma}(\sigma(\vec{p}))$ on the inputs and check whether $\hat{\sigma}(\sigma(\vec{p}))(\vec{\imath}) = \vec{o}$.

For some instances, we explicitly construct the problem such that there exists only a single unique solution, in which case the inverse permutation $\hat{\sigma} = \sigma^{-1}$ is the only correct answer. The evaluation metric used is accuracy (Acc), defined analogously to `Pass@1`:

$$\text{Acc} = \frac{1}{|\mathcal{D}|} \sum_{\vec{o},\vec{\imath} \in \mathcal{D}} 1_{\hat{\sigma}(\sigma(\vec{p}))(\vec{\imath})=\vec{o}}$$

We additionally report unique accuracy, defined as the accuracy computed over the subset of instances with a single valid solution. In the PBEBench-Lite-Perm dataset, this subset consists of 242 instances.

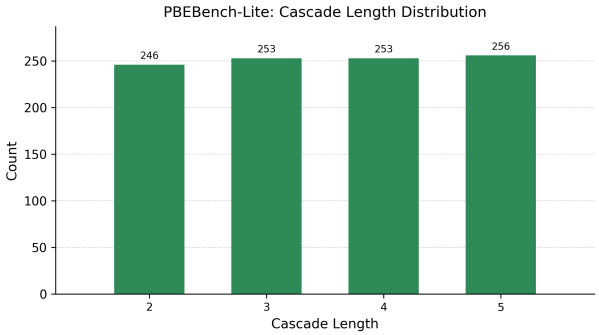


Figure 3: Cascade length distribution for PBEBench-Lite.

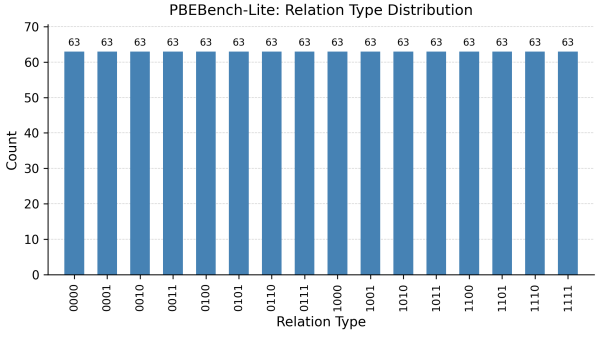


Figure 4: Relation type distribution for PBEBench-Lite.

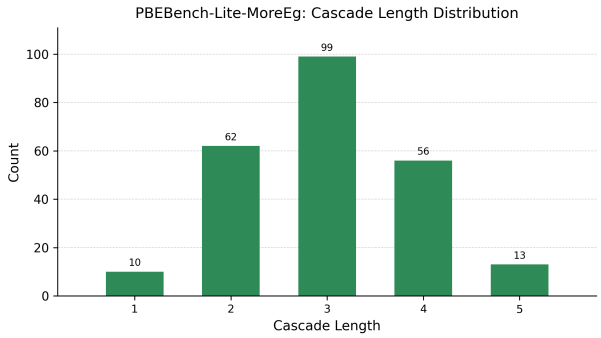


Figure 5: Cascade length distribution for PBEBench-Lite-MoreEg.

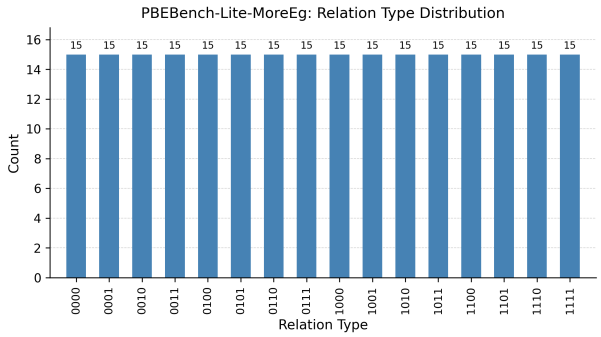


Figure 6: Relation type distribution for PBEBench-Lite-MoreEg.

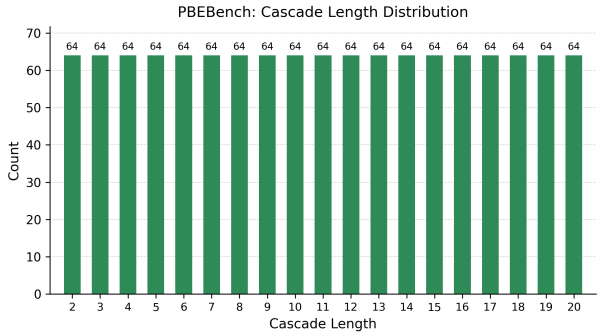


Figure 7: Cascade length distribution for PBEBench.

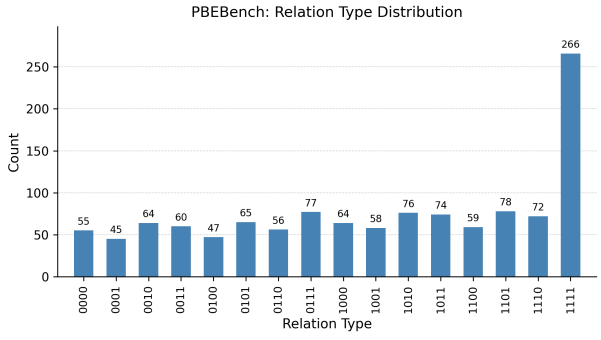


Figure 8: Relation type distribution for PBEBench.

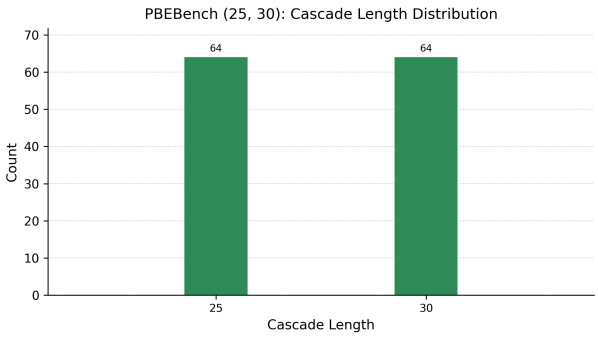


Figure 9: Cascade length distribution for PBEBench (25, 30).

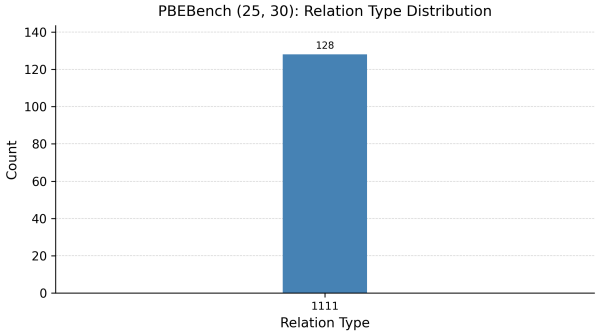


Figure 10: Relation type distribution for PBEBench (25, 30).

D Method Details

This section provides additional details on the problem proposer and problem solver to support reproducibility. Specifically, it covers:

- **Problem proposer:** the algorithm used for data generation.
- **Problem solver:** the prompt template for inference, the models used in our work (including details, licensing, and snapshots for closed-source models, as well as costs of running them), and the K-Fold analysis employed to efficiently simulate different sampling budgets with expensive closed-source models like GPT-5 while reducing variance.

D.1 Algorithm Pseudocode

This section formally expresses the pseudocode/-logic behind the data generation algorithm proposed in our work, breaking it down into the rejection sampling subroutine (Algorithm 1) and the relation type classification subroutine (Algorithm 2), respectively.

D.2 Program Extraction

To extract the program cascade from LLM predictions, we apply the following rules. If the LLM fails to produce a valid, extractable Python code block, or produces a null response due to unterminated chain of thought (as observed in gpt-oss models), we mark the corresponding case as a null response and replace the prediction with an identity program like `replace("x", "x")` for evaluation. For cases where a valid response is produced, we account for the fact that we evaluate a wide range of LLMs, including reasoning models that may generate intermediate programs and subsequently refine or improve them through reflection. Accordingly, we evaluate both the first and the last Python code blocks produced by the model. For the remaining valid cases, if a predicted program cascade $\hat{\vec{p}}$ contains more than L_{max} programs, we retain only the first L_{max} programs for evaluation. If the predicted program violates any other constraint, it is replaced by an identity program p^I, which leaves the inputs unchanged. Given the predicted cascade $\hat{\vec{p}}$, we compute the predicted outputs as $\hat{\vec{o}} = \hat{\vec{p}}(\vec{i})$. Table 3 reports performance on PBEBench-Lite for both the first and last code blocks. We observe that, for most models, the last code block yields superior performance. Therefore, in Table 2 in the main paper, as well as in any table where it is not explicitly specified otherwise, we report results corresponding to the last code block by default.

Algorithm 1 Rejection Sampling for BFCC Dataset Generation

Require: Target size n; cascade bounds $[\ell_{\min}, \ell_{\max}]$; patience τ
Require: Input sampler $p_{\mathcal{I}}$: generates k random strings from vocabulary Σ
Require: Cascade sampler $p_{\mathcal{P}}(\cdot \mid X, \ell)$: generates ℓ programs where each replace(a, b) has a sampled from substrings of X and b sampled from Σ
Ensure: Dataset $\mathcal{D}$ balanced over 16 BFCC categories

Initialize quotas $q_c \leftarrow \lfloor n/16 \rfloor$ for all $c \in \{0,1\}^4$
Initialize $\mathcal{D} \leftarrow \emptyset$, seen signatures $\mathcal{S} \leftarrow \emptyset$, steps $t \leftarrow 0$
while $|\mathcal{D}| < n$ **do**
  $t \leftarrow t + 1$
  Sample length $\ell \sim$ Uniform$\{\ell_{\min}, \ldots, \ell_{\max}\}$
  Sample inputs $X = \{x_1, \ldots, x_k\} \sim p_{\mathcal{I}}$
  Sample cascade $\pi = (f_1, \ldots, f_\ell) \sim p_{\mathcal{P}}(\cdot \mid X, \ell)$
  $\hat{\pi} \leftarrow []$; $Y \leftarrow X$
  for each program $f \in \pi$ **do**
    if f changes at least one element in Y **then**
      $\hat{\pi} \leftarrow \hat{\pi} \cdot f$; $Y \leftarrow f(Y)$
    end if
  end for
  if $|\hat{\pi}| < \ell_{\min}$ or $Y = X$ **then**
    continue ▷ Reject: insufficient transformation
  end if
  $\sigma \leftarrow (X, Y, \hat{\pi}, |\hat{\pi}|)$
  if $\sigma \in \mathcal{S}$ **then**
    continue ▷ Reject: duplicate
  end if
  $c \leftarrow$ ClassifyBFCC$(\hat{\pi})$
  if $t < \tau$ and $q_c > 0$ **then** ▷ Before patience: enforce quotas
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(X, \hat{\pi}, Y, c)\}$
    $q_c \leftarrow q_c - 1$; $\mathcal{S} \leftarrow \mathcal{S} \cup \{\sigma\}$
  else if $t \geq \tau$ **then** ▷ After patience: accept any valid instance
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(X, \hat{\pi}, Y, c)\}$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{\sigma\}$
  end if
end while
return $\mathcal{D}$

Model	First Code Block			Last Code Block		
	Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate
Codestral-22B	0.0109	-0.0130	0.8280	0.0109	-0.0107	0.8254
Qwen2.5-32B-Instruct	0.0298	0.1232	0.8265	0.0298	0.1252	0.8288
Qwen2.5Coder-32B-Instruct	0.0397	0.1836	0.6883	0.0407	0.1884	0.6890
Qwen3-32B	0.0179	0.0964	0.7645	0.0179	0.0964	0.7645
Qwen3-Coder-30B-A3B-Instruct ■	0.0377	0.0857	0.8048	0.0377	0.0864	0.8134
DeepSeek-R1-Distill-Qwen-32B ★	0.2242	0.3486	0.8709	0.2242	0.3486	0.8709
Qwen3-30B-A3B ★■	0.2887	0.3360	**0.9905**	0.2887	0.3360	**0.9905**
QwQ-32B ★	0.3591	0.4072	0.9500	0.3601	0.4092	0.9493
Qwen3-32B (with CoT) ★	0.3938	0.4803	0.9182	0.4187	0.5026	0.9676
gpt-oss-20b ★■	0.4058	0.4619	0.9900	0.4058	0.4619	0.9900
gpt-oss-120b ★■	**0.6250**	**0.6985**	0.9254	**0.6250**	**0.6985**	0.9254
Claude-3.5-Sonnet	0.1845	0.4430	0.8208	0.1845	0.4434	0.8209
Claude-3.7-Sonnet	0.2212	0.4825	0.8409	0.2321	0.4996	0.8409
Claude-4 Sonnet	0.2956	0.5832	0.7720	0.2966	0.5870	0.7719
Claude-3.7-Sonnet (Thinking) ★	0.3343	0.5953	0.8127	0.3661	0.6154	0.8192
Claude-4 Sonnet (Thinking) ★	0.3571	0.6085	0.7788	0.3581	0.6079	0.7821
Claude-4 Opus (Thinking)* ★	0.5389	0.7497	0.8577	0.5389	0.7521	0.8561
o3-mini ★	0.5278	0.5954	0.9179	0.5278	0.5954	0.9179
o4-mini ★	0.6329	0.6907	0.9153	0.6329	0.6907	0.9153
Gemini 2.5 Flash ★	0.5833	0.6533	0.7911	0.5863	0.6562	0.7901
GPT-5 ★	**0.7242**	**0.7645**	0.9286	**0.7242**	**0.7645**	0.9286

Table 3: **PBEBench-Lite Performance:** We compute the `Pass@1` and `Edit_Sim` as the coarse and fine-grained evaluation, respectively, for each model. ■ indicates mixture-of-experts (or MoE) model. ★ indicates a reasoning model. * indicates evaluated on 20% of the dataset due to cost.

Algorithm 2 BFCC Classification

function CLASSIFYBFCC(cascade $\hat{\pi} = [\text{replace}(a_1, b_1), \ldots, \text{replace}(a_m, b_m)]$)
  Initialize category vector $(c_F, c_B, c_{CF}, c_{CB}) \leftarrow (0, 0, 0, 0)$
  for each ordered pair (i, j) where $i \neq j$ **do**
    if program i feeds program j **then** ▷ b_i creates instances of a_j
      Set $c_F \leftarrow 1$ if $i < j$; set $c_{CF} \leftarrow 1$ if $i > j$
    end if
    if program i bleeds program j **then** ▷ a_i removes instances of a_j
      Set $c_B \leftarrow 1$ if $i < j$; set $c_{CB} \leftarrow 1$ if $i > j$
    end if
  end for
  return $(c_F, c_B, c_{CF}, c_{CB})$
end function

Algorithm 3 FB-Swap Permutation Heuristic

FB. "FB" refers to *feeding* (F) and *bleeding* (B) relations between programs (Algorithm 2).
Execution. APPLYPROGRAMS(X, π) applies the programs in π left-to-right to each string in X.

function FINDFBSWAP($X, \hat{\pi}, Y, E_{FB}$)
Require: Cascade $\hat{\pi} = [f_0, \ldots, f_{m-1}]$ and outputs $Y =$ APPLYPROGRAMS$(X, \hat{\pi})$
Require: FB edges E_{FB} is a list of triples (i, r, j) with $r \in \{F, B\}$
Ensure: A permutation ρ and permuted cascade $\hat{\pi}^{\rho}$ such that APPLYPROGRAMS$(X, \hat{\pi}^{\rho}) \neq Y$, or $\varnothing$ if none found
    $m \leftarrow |\hat{\pi}|$
    $S \leftarrow \emptyset$ ▷ unordered index pairs already tried
    for each edge $(i, r, j) \in E_{FB}$ **do**
        $p \leftarrow (\min(i, j), \max(i, j))$
        if $p \in S$ **then**
            continue ▷ avoid trying the same swap twice
        end if
        $S \leftarrow S \cup \{p\}$
        $\rho \leftarrow [0, 1, \ldots, m-1]$ ▷ identity permutation over program positions
        swap ρ_i and ρ_j ▷ single transposition
        $\hat{\pi}^{\rho} \leftarrow [f_{\rho_0}, \ldots, f_{\rho_{m-1}}]$
        if APPLYPROGRAMS$(X, \hat{\pi}^{\rho}) \neq Y$ **then**
            return $(\rho, \hat{\pi}^{\rho})$ ▷ first FB-related swap that changes outputs
        end if
    end for
    return $\varnothing$
end function

Multi-Step PBE Prompt

Follow the instructions below to solve the code completion task:
We will provide the input corpus and corresponding output corpus. Each element in the corpus is a string, and the output is transformed from the corresponding input using an ordered sequence of "replace" programs. You need to find the correctly constructed and ordered sequence of "replace" programs to transform the entire input corpus into the output corpus. Note that the programs can interact with each other in a way that reduces or increases the number of times they are applied on a given input based on where they are ordered in the sequence. This makes it very important to apply them in the correct order.
The programs should be written using only the Python `replace` function. For example, for a program that replaces all occurrences of "ab" with "bc" it should be written as: `replace('ab', 'bc')`
Here is an example of the full task:

```
### Inputs
["abc", "ebc", "aba"]

### Outputs
["edc", "edc", "aba"]

### Program Sequence
```python
["replace('bc','dc')", "replace('ad','ed')"]
```
```

While generating the program sequence, you need to abide by the following restrictions:

1. Each program in the sequence should have the form `replace(A, B)`, where `A` and `B` are both strings.
2. Both argument strings `A` and `B` in `replace(A, B)` should have length $\leq \{program\_length\}$. `A` must have length ≥ 1, while `B` may be empty (i.e., `""`).
3. The maximum number of programs in a sequence is {program_num}.
4. You should only consider the Python `replace` function for specifying programs (each program is a Python `replace` function). You cannot use any other Python modules or functions.
5. Strictly follow the markdown style convention while presenting your final program sequence, and make sure to enclose it in the ```` ```python ```` markdown style code block.

Now, please generate the sequence of programs corresponding to the following input corpus and output corpus:

```
### Inputs
{inputs_list}

### Outputs
{outputs_list}

### Program Sequence
```

D.3 Prompt Templates

D.3.1 Multi-step PBE Task

We show the prompt template used for the multi-step PBE task above (`Multi-Step PBE Prompt`). This prompt includes the exact instructions and examples given to the LLMs.

D.3.2 Program Reordering Task

We show the prompt template used for the program reodering task below (`Program Reordering Prompt`). This prompt includes the exact instructions and examples given to all the LLMs for performing this task.

D.4 Model Selection Details

Table 4 details all the models chosen for our benchmark and their various attributes to ensure we evaluate a diverse and representative set of LLMs to evaluate which of them excel at inductive reasoning.

D.5 Licenses for Evaluated Models

We list the licenses used for each evaluated open and closed source models in Table 5.

D.6 Costs for Experiment Runs

We document the costs of the expensive experiments carried out for closed source models in Table 6.

D.7 Snapshots used for closed source models

We document the Snapshots used for closed source models Table 7.

D.8 K-Fold analysis for GPT-5

We observed a variance of up to 10% in Pass@1 for GPT-5. To account for this variance, we report the aggregated score over all available samples when computing Pass@1 values. For instance, in the sampling experiment for GPT-5 shown in Fig. 12a, we perform 8 independent runs and compute the average score over all possible k-run combinations, yielding scores for sampling budgets $1 \leq k \leq 8$.

E Experimental Details

In this section, we provide additional experimental details, including the computational environment and the inference and sampling parameters used for all the LLMs evaluated in our work. We also briefly discuss the strategies explored to achieve finer-grained control over the thinking budget of gpt-oss-120b for scaling experiments. These strategies were ultimately unsuccessful due to gpt-oss-120b's test-time behavior, leading us to instead study the effect of varying the maximum sequence length directly.

E.1 Computational Enviornment

We conduct experiments on a Linux server equipped with NVIDIA A100 80GB GPUs (Ampere architecture), CUDA 12.9, and driver version 575.51.03. Each job had access to 100 GB of CPU memory and up to 16 CPU cores. The GPU allocation varied with model size with gpt-oss-120b and most 32B models requiring 2 A100 GPUs. The experiments on PBEBench-Lite took multiple hours, while each cascade on PBEBench took nearly 8 hours for sampling budget of 32 and 16384 max sequence length, prompting use to parallely run multiple cascades across several GPUs. We used vLLM for inference of all the open weight models and multi-threading for inference of closed source models like GPT-5 to speed up all inference experiments.

E.2 Inference/Sampling Parameters

We show the sampling parameters used for all the models in Table 8. The max tokens are the total output tokens the model can generate (including thinking tokens), while the thinking budget(s) captures only the chain-of-thought or reasoning related tokens. The top-p is the cumulative probability cutoff used for nucleus sampling, while the temperature is for controlling the degree of randomness in the sampling. We report the max tokens and thinking tokens wherever possible based on the providers (for some models you can only control the total tokens, while for some you can only control thinking tokens). For some models like Gemini 2.5 Flash Preview, the model has a mode where it first reasons about how much thinking is required based on how complex it determines the problem to be. We use this setting for the experiments in Table 2. However, we also do experiments comparing the effect of varying token budgets (2048, 4096, 8192) for QwQ and Gemini 2.5 Flash Preview, hence we highlight the default setting used for Table 2 for these models in bold.

E.3 CoT Truncation experiment details

For gpt-oss-120b, we attempt to reduce the model's *thinking budget* and introduce it as a parameter

Model Name	Reasoning	Citation	Parameters	MoE	Source
QwQ-32B	Yes	Team (2025b)	32B	No	Closed
DeepSeek-R1-Distill-Qwen-32B	Yes	DeepSeek-AI (2025)	32B	No	Closed
o3-mini	Yes	OpenAI (2024)	–	No	Closed
o4-mini	Yes	OpenAI (2025)	–	No	Closed
Qwen3-30B-A3B (Thinking)	Yes	Team (2025a)	30B	Yes	Open
Qwen3-32B	Yes	Team (2025a)	32B	No	Open
Gemini 2.5 Flash	Yes	Comanici et al. (2025)	–	No	Closed
Claude-3.7-Sonnet	Yes	Anthropic (2025)	–	No	Closed
Claude-4 Sonnet (Thinking)	Yes	Anthropic (2025b)	–	No	Closed
Claude-4 Opus (Thinking)	Yes	Anthropic (2025a)	–	No	Closed
gpt-oss-20b	Yes	OpenAI (2025)	20B	No	Open
gpt-oss-120b	Yes	OpenAI (2025)	120B	No	Open
GPT-5 (Thinking)	Yes	OpenAI (2025)	–	No	Closed
Qwen2.5-32B-Instruct	No	Team (2024)	32B	No	Open
Claude-3.5-Sonnet	No	Anthropic (2024)	–	No	Closed
GPT-5 (Non-Thinking)	No	OpenAI (2025)	–	No	Closed
Codestral-22B	No	AI (2024)	22B	No	Open
Qwen2.5Coder-32B-Instruct	No	Team (2024)	32B	No	Open
Qwen3-Coder-30B-A3B-Instruct	No	Team (2025a)	30B	Yes	Open

Table 4: **Model Selection:** This table details the characteristics of the models benchmarked on PBEBench-Lite. The columns discuss cover the model name, reasoning ability, citation, parameter count, architecture style (MoE vs Dense), and open/closed soruce nature of the chosen models to showcase the diversity of the evaluated models.

Model	**License**
Codestral-22B	Mistral Non-Production License (MNPL)
Qwen2.5-32B-Instruct	Apache 2.0
Qwen2.5Coder-32B-Instruct	Apache 2.0
Qwen3-32B	Apache 2.0
Qwen3-Coder-30B-A3B-Instruct	Apache 2.0
QwQ-32B	Apache 2.0
Qwen3-32B	Apache 2.0
Qwen3-30B-A3B	Apache 2.0
Qwen3-Coder-30B-A3B-Instruct	Apache 2.0
DeepSeek-R1-Distill-Qwen-32B	MIT
o3-mini	API (OpenAI EULA)
o4-mini	API (OpenAI EULA)
GPT-5	API (OpenAI EULA)
gpt-oss-20b	Apache 2.0
gpt-oss-120b	Apache 2.0
Gemini 2.5 Flash Preview 04-17	API (Google EULA)
Claude-3.5-Sonnet	API (Anthropic EULA)
Claude-3.7-Sonnet	API (Anthropic EULA)
Claude-4-Sonnet	API (Anthropic EULA)
Claude-4-Opus	API (Anthropic EULA)

Table 5: Licenses for open and closed source models.

independent of *max tokens*. To achieve this, we run two inferences:

Model	Experiment	Cost
Claude-4.1-Opus	PBE-Bench Lite Performance	$40 (20% of dataset)
GPT-5	Cascade Length Experiment	$190
GPT-5	Sampling Experiment	$165
GPT-5	CoT Experiment	$50
GPT-5	PBE-Bench Lite Performance	$50
Claude Sonnet Thinking 3.7	PBE-Bench Lite Performance	$30
Claude Sonnet Thinking 4	PBE-Bench Lite Performance	$30
o3-mini	PBE-Bench Lite Performance	$65
o4-mini	PBE-Bench Lite Performance	$65

Table 6: Documented costs for select experiment runs.

Model	Snapshot
o3-mini	o3-mini-2025-01-31
o4-mini	o4-mini-2025-04-16
Claude 3.5 Sonnet	claude-3-5-sonnet-20241022
Claude 3.7 Sonnet	claude-3-7-sonnet-20250219
Claude 4 Sonnet	claude-sonnet-4-20250514
Claude 4.1 Opus	claude-opus-4-1-20250805
GPT-5	gpt-5-2025-08-07
Gemini 2.5	gemini-2.5-flash

Table 7: Exact snapshots used for closed source models.

1. In the first pass, we set *max tokens* equal to the desired thinking budget. We then check whether the Chain-of-Thought Truncation token appears in the response.

2. If it does not appear, we run a second inference, appending the following string as assistant context to the model's input:

```
early_stop_instruction = "Considering the thinking token budget, I will not generate any more reasoning tokens, and provide the final answer `.\n"
```

In the second generation, however, the model begins with "We need to produce final answer" and then continues reasoning as usual, ignoring our instruction.

In variations, we appended THINKING_END_TOKEN, and <FINAL_OUTPUT_START_TOKEN> to the *early_stop_instruction*, but observed the same behavior. We also tried placing a modified version of *early_stop_instruction* in the *user* role instead of the assistant role, again without effect. Finally, we reduced *max tokens* to 300 in the second generation. This did not prompt the model to produce a final answer either; instead, It significantly increased the rate of null outputs, rising from 11% to 77%. We therefore conclude that for gpt-oss-120b, it is not possible to enforce the truncation of the chain-of-thought budget independently of the total output token budget.

F Additional Results

This section presents some additional detailed results over the PBEBench-Lite and PBEBench, and PBEBench (25, 30) snapshots, such as the detailed tables for the performance vs ground truth cascade length, scaling ablations, etc. We also report results on related inductive reasoning benchmarks as well as on real SLI data. It also contains the results of analyzing the effect of changing the number of examples (PBEBench-Lite-MoreEg snapshot). It also contains the results of logistic regression analysis on factors affecting instance difficulty on PBEBench with gpt-oss-120b and factorial analysis on reasoning-capable models, as well as some representative models on PBEBench-Lite. Finally, we also present results for two types of confusion matrices that visualize the distributional differences between cascade lengths and relation types of the

Model	Max Tokens	Top P	Temperature	Thinking Budget(s)
Codestral-22B	2048	0.95	0.7	-
Qwen2.5-32B-Instruct	512	0.95	0.7	-
Qwen2.5Coder-32B-Instruct	512	0.95	0.7	-
QwQ-32B	8192	0.95	0.7	-
Qwen/Qwen3-32B (with CoT)	8192	0.95	0.7	-
Qwen/Qwen3-32B	8192	0.95	0.7	-
Qwen3-30B-A3B	8192	0.95	0.7	-
DeepSeek-R1-Distill-Qwen-32B	8192	0.95	0.7	-
o3-mini	8192	-	-	reasoning_effort="medium"
o4-mini	8192	-	-	reasoning_effort="medium"
Gemini 2.5 Flash	dynamic	0.95	0.7	dynamic
Claude-3.5-Sonnet	8192	0.95	0.7	-
Claude-3.7-Sonnet	10000	0.95	0.7	-
Claude-3.7-Sonnet (Thinking)	10000	0.95	1 (default)	2048
gpt-oss-20b	8192	0.95	0.7	-
gpt-oss-120b	8192	0.95	0.7	-
GPT-5	8192	-	-	reasoning_effort="medium"
Claude-4 sonnet	8192	0.95	0.7	-
Claude-4 sonnet (Thinking)	8192	0.95	1 (default)	2048
Claude-4 opus (Thinking) (20% of dataset)	8192	0.95	1 (default)	2048
Qwen/Qwen3-Coder-30B-A3B-Instruct	2048	0.95	0.7	-

Table 8: Sampling parameters used for inference across all models runs. “Max tokens” refers to the total number of tokens (output + thinking tokens) for models that support it. "Top-p" controls nucleus sampling. "Temperature" sets the randomness of token selection. "Thinking budget" is the number of thinking tokens, applicable only to models that support this feature. GPT-5, o3-mini, and o4-mini support "reasoning_effort" parameter, which is a qualitative measure of "Thinking Budget". These models also do not support temperature and top_p parameters.

ground truth and predicted cascades.

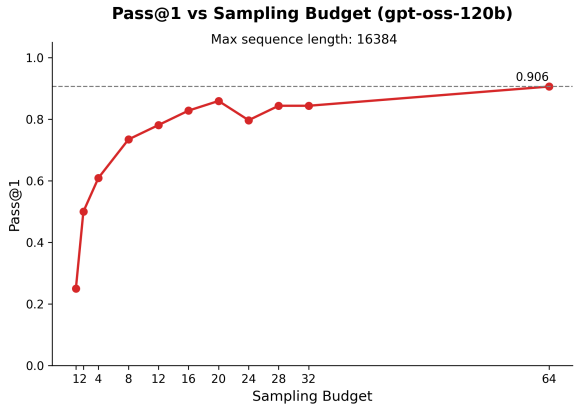
F.1 PBEBench Performance Details

We evaluate gpt-oss-120b and GPT-5 on PBEBench, which includes cascades of length 2–20 and harder snapshots of length 25 and 30. On cascades of length 2–20, gpt-oss-120b achieves `Pass@1` of 0.67, average `Edit_Sim` of 0.95, and `Valid_Rate` of 0.96, with per-cascade `Pass@1` reported in Table 20. GPT-5 substantially outperforms gpt-oss-120b, achieving `Pass@1` of 0.94 on cascades of length 10 with a single sample (vs. 0.84 with 32 samples for gpt-oss-120b) and 0.44 on cascades of length 20 with 4 samples (vs. 0.05 with 32 samples). Since GPT-5 maintains meaningful performance at length 20, unlike gpt-oss-120b which collapses to 5%, we further evaluate GPT-5 on cascades of length 20, 25, and 30, where it averages `Pass@1` of 0.175 (Table 22). Performance trends are shown in Fig. 2a and Fig. 2b.

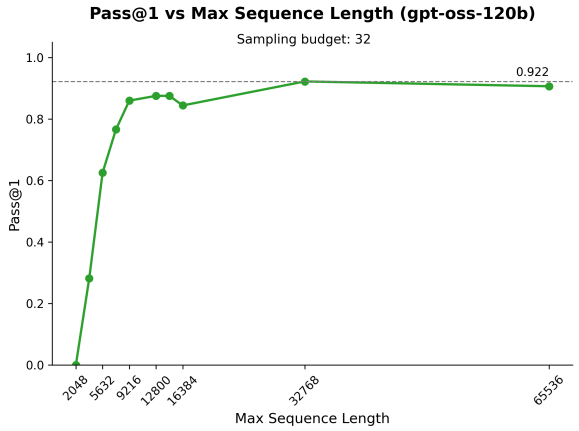
Both models follow the scaling strategies described in Section 3.2. gpt-oss-120b uses a sampling budget of 32 with maximum sequence length 16384, while GPT-5 uses a sampling budget of 4, a completion limit of 65536 tokens, and moderate reasoning effort. Performance drops sharply with increasing cascade length: gpt-oss-120b falls to roughly 50% and 5% at lengths 15 and 20, while GPT-5 falls to 14% and 5% at lengths 25 and 30. A logistic regression on gpt-oss-120b identifies cascade length as a strong negative predictor of `Pass@1`, with feeding, counter-feeding, and counter-bleeding predicting failure, and bleeding associated with success (Table 19).

F.2 Ablation Details

We evaluate gpt-oss-20b and gpt-oss-120b on the PBEBench-Lite-MoreEg snapshot, with results reported in Table 21. Performance is similar to or slightly worse than PBEBench-Lite. To analyze this effect, we compute the fraction of examples modified by each program: 1.56/5 (31%) for PBEBench-Lite and 7.07/50 (14%) for PBEBench-Lite-MoreEg. Although more examples are modified in absolute terms, the relative signal per example is weaker. However, the overall conclusion is that the effect of scaling input-output examples per PBE step from 5 to 50 (10x) is still relatively minor. However, after a certain point, the LLM is expected

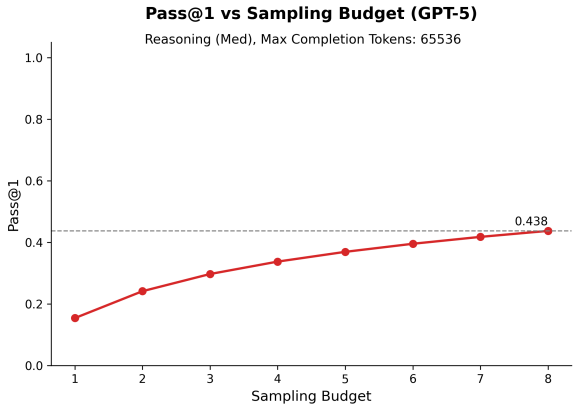


(a) Pass@1 vs Sampling Budget (max seq len: 16384)

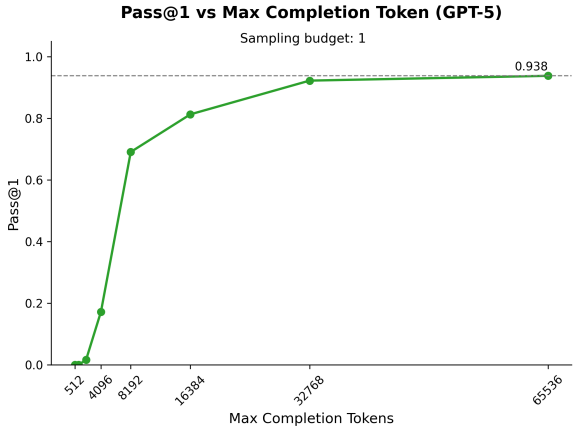


(b) Pass@1 vs Max Seq Len (sampling budget: 32)

Figure 11: **Effect of Scaling Strategies on PBEBench (gpt-oss-120b):** Comparison of successive sampling and increased thinking budgets (via max sequence length) on PBEBench instances with ground-truth cascades of length 10, the most complex balanced subset, unsolved yet nearly solvable under greater scaling, allowing a meaningful strategy comparison.



(a) Pass@1 vs Sampling Budget (reasoning (med), max completion tokens: 65536)



(b) Pass@1 vs Max Completion Tokens (sampling budget: 1)

Figure 12: **Effect of Scaling Strategies on PBEBench (GPT-5):** Comparison of successive sampling and larger thinking budgets (via max completion tokens and reasoning effort) on PBEBench with ground-truth cascade lengths 20 and 10 respectively. Cascade 20 is used for the sampling budget experiment as we observe high `Pass@1` (0.938) for cascade 10 with just 1 sample, in the max completion tokens scaling experiment.

to hit a hard wall, as experiments with real SLI data, where we attempted to have 20 examples per PBE step, led to models like gpt-oss-120b giving up and providing empty responses.

We further study scaling with respect to sampling budget and maximum sequence length. For gpt-oss-120b, results are shown in Fig. 11a and Fig. 11b. For GPT-5, corresponding results are shown in Fig. 12a and Fig. 12b, with k-fold averaging used to control variance (Section D.8). Across all settings, we observe rapid initial gains followed by diminishing returns and eventual saturation. Interestingly, for the more head-on comparison done for gpt-oss-120b, we see that scaling the max sequence length (or thinking budget) once you have a high enough sampling budget is more time efficient and cheaper. This stems from the fact that a high sampling budget is more wasteful if the LLM is token-bound per attempt, as a lot of attempts fail to terminate the chain-of-thought and lead to unhelpful null responses.

F.3 Other Inductive Reasoning Benchmarks

We evaluate all open-source models on two additional inductive reasoning benchmarks: CLUTRR (Sinha et al., 2019) and SLR-Bench (Helff et al., 2025).

CLUTRR is a well-established NLU benchmark that requires LLMs to read a short story, extract relationships between characters, and answer a kinship query about a target pair. The answer space consists of a small, fixed set of kinship relations, making the task a multi-class classification prob-

lem. Successfully solving CLUTRR requires both accurate extraction of relational facts and correct inference over the underlying logical rules governing kinship. The dataset is constructed semi-automatically from a kinship knowledge base via four steps: random kinship graph generation, target fact (relation) sampling, backward chaining, and natural language realization. CLUTRR also explicitly probes robustness and generalization by injecting distracting facts into the stories, including supporting, irrelevant, disconnected, and noisy facts. Following prior work, we concatenate all six test sets introduced in the original paper to form a combined evaluation set of 3,977 instances. While the original CLUTRR work involves supervised training, we instead evaluate models in a zero-shot question answering setting using a fixed prompt (`CLUTRR Prompt`). Results on the combined test set are reported in Table 9. We report standard kinship prediction accuracy, along with additional metrics for gpt-oss models, including non-null accuracy and the number of null predictions. Null outputs arise when the chain of thought fails to terminate within the specified maximum sequence length, typically due to overthinking. To further analyze this behavior, we vary both the reasoning effort and maximum sequence length for these models and report the corresponding performance. Overall, LLMs perform poorly on CLUTRR, likely due to the presence of distractor facts and the absence of training or explicit graph-based representations for tracking relations. In contrast to PBEBench, however, performance differences across models are relatively small, with a narrower gap between reasoning and non-reasoning models. This trend is also reflected in the limited performance variation across different reasoning effort settings for gpt-oss models, and in the degradation observed at high reasoning effort, where increased null predictions from unterminated chains of thought reduce accuracy. Finally, to ensure comparability with PBEBench-Lite, we restrict each model to a single attempt per instance.

SLR-Bench is a large-scale, automatically generated benchmark for logical inductive reasoning, constructed using a fully automated framework that synthesizes prompts, validation programs, and latent rules without any human annotation. It comprises 19k tasks organized into a curriculum of increasing relational, arithmetic, and recursive complexity, enabling fine-grained evaluation of logical inference capabilities in LLMs. The benchmark includes validation programs, and the solutions produced by the LLMs are expressed as Prolog code, which is subsequently evaluated by a symbolic judge that executes the validation program over the LLM generated solution. In the original work, the curriculum is used for training and is shown to provide benefits on the SLR-Bench test set as well as on other general reasoning benchmarks. However, to remain consistent with our experimental setting, we perform a purely zero-shot evaluation on the combined test set, which includes the basic, easy, medium, and hard tiers, yielding a dataset of 1000 instances. We report all metrics introduced by the original authors, including accuracy (`Acc`), partial score (`PS`), and syntax score (`SS`), which correspond respectively to the fraction of instances that are fully solved, the average fraction of examples correctly classified (analogous to test cases passed), and the fraction of instances for which a syntactically valid Prolog program is produced. We use the exact prompts provided in the dataset, rather than creating our own templates, in order to ensure faithful and directly comparable evaluation. The aggregate results are shown in Table 11, while a breakdown by curriculum tier is provided in Table 12. Overall, the results follow a trend similar to that observed on CLUTRR, where reasoning-oriented models generally outperform non-reasoning models, although the performance gap is smaller than CLUTRR. Among all evaluated models, `gpt-oss-120b` with medium reasoning effort achieves the best performance, reaching 52.5% accuracy. Also consistent with CLUTRR, we observe that performance for both `gpt-oss` models peaks at medium reasoning effort and subsequently declines, primarily due to an increase in null outputs caused by unterminated chains-of-thought associated with overthinking. A key complicating factor relative to CLUTRR and PBEBench-Lite is the inherent difficulty of producing syntactically valid Prolog programs. Several models, including `QwQ-32B`, `Qwen3-32B`, and `gpt-oss-20b` with medium reasoning effort, achieve overall scores around 50%, indicating substantial difficulty in generating syntactically correct code; in the case of `gpt-oss-20b`, this behavior is largely explained by a higher proportion of null outputs. In contrast, coder-oriented models such as `Codestral-22B` and `Qwen2.5-Coder-32B-Instruct`, as well as `Qwen2.5-32B-Instruct`, excel at producing syntactically valid programs, achieving nearly 100% syntax scores despite comparatively weak overall

CLUTRR Prompt

You will be given a story containing characters whose names appear in square brackets, such as "[James]". After reading the story, you must identify the kinship relation between two characters.

When answering:

- Respond with only the kinship term, with no explanation and no extra words.
- Your answer must be one of the following options: aunt, brother, daughter, daughter-in-law, father, father-in-law, granddaughter, grandfather, grandmother, grandson, mother, mother-in-law, nephew, niece, sister, son, son-in-law, uncle.
- Do not use any text outside these options.
- Give just the final relation.

Example Story: [Kristin] and her son [Justin] went to visit her mother [Carol] on a nice Sunday afternoon. They went out for a movie together and had a good time.

Example Question: How is Carol related to Justin?

Example Answer: grandmother

Now do the same for the story and question below:

Story: `{story}`

Question: `{question}`

Answer:

Model	Max Seq Len	Acc	Non Null Acc	Nulls
Codestral-22B	2048	3.72	3.72	0
Qwen/Qwen2.5-32B-Instruct	8192	34.98	34.98	0
Qwen/Qwen2.5-Coder-32B-Instruct	8192	26.88	26.88	0
Qwen/QwQ-32B	8192	52.07	52.07	0
Qwen/Qwen3-32B (with CoT)	8192	51.62	51.62	0
Qwen/Qwen3-32B	8192	22.53	22.53	0
Qwen/Qwen3-30B-A3B	8192	51.47	51.47	0
DeepSeek-R1-Distill-Qwen-32B	8192	47.67	47.67	0
gpt-oss-20B (low)	8192	42.59	42.59	0
gpt-oss-20B (medium)	8192	52.5	54.53	148
gpt-oss-20B (high)	8192	45.64	60.76	990
	16384	50.64	60.04	621
	32768	53.1	58.46	330
gpt-oss-120B (low)	8192	47.55	47.55	0
gpt-oss-120B (medium)	8192	57.53	57.53	0
gpt-oss-120B (high)	8192	51.72	63.43	631
	16384	57.98	61.81	246
Qwen/Qwen3-Coder-30B-A3B-Instruct	8192	26.4	26.4	0

Table 9: **CLUTRR Performance:** We compute the kinship relation type prediction accuracy (`Acc`) as well as non-null accuracy (`Non Null Acc`) which excludes null predictions and the number of null predictions (`Nulls`). We also try increased max sequence length and vary reasoning effort for the gpt-oss models.

Model	Max Seq Len	Gen Train234		Gen Train23		Rob Clean		Rob Disc		Rob Irr		Rob Sup	
		Acc	NNAcc	Acc	NNAcc	Acc	NNAcc	Acc	NNAcc	Acc	NNAcc	Acc	NNAcc
Codestral-22B	2048	3.72	3.72	4.71	4.71	0.22	0.22	2.25	2.25	3.6	3.6	6.26	6.26
Qwen/Qwen2.5-32B-Instruct	8192	42.08	42.08	45.03	45.03	21.25	21.25	25.39	25.39	23.42	23.42	27.29	27.29
Qwen/Qwen2.5-Coder-32B-Instruct	8192	35.02	35.02	36.47	36.47	14.54	14.54	19.33	19.33	11.26	11.26	18.57	18.57
Qwen/QwQ-32B	8192	61.26	61.26	63.61	63.61	41.39	41.39	38.88	38.88	37.61	37.61	39.15	39.15
Qwen/Qwen3-32B (with CoT)	8192	60.78	60.78	63.26	63.26	37.81	37.81	41.12	41.12	37.39	37.39	38.7	38.7
Qwen/Qwen3-32B	8192	27.1	27.1	27.4	27.4	13.87	13.87	15.06	15.06	16.67	16.67	21.25	21.25
Qwen/Qwen3-30B-A3B	8192	61.07	61.07	61.95	61.95	37.58	37.58	41.12	41.12	37.84	37.84	39.82	39.82
DeepSeek-R1-Distill-Qwen-32B	8192	55.92	55.92	59.16	59.16	36.02	36.02	35.51	35.51	34.23	34.23	36.02	36.02
gpt-oss-20B (low)	8192	46.09	46.09	46.95	46.95	36.69	36.69	41.8	41.8	36.04	36.04	36.47	36.47
gpt-oss-20B (medium)	8192	61.55	65.68	62.04	66.51	40.27	40.27	44.49	44.59	39.19	39.46	40.27	40.36
	8192	48.28	77.73	54.36	80.08	40.27	45.11	41.12	49.06	35.59	40.51	36.91	41.67
gpt-oss-20B (high)	16384	56.49	76.19	59.77	77.14	44.07	46.03	45.39	48.56	36.4	38.28	40	41.72
	32768	61.35	71.6	63.53	73.17	44.07	44.57	46.52	47.37	39.19	40	40.94	41.59
gpt-oss-120B (low)	8192	50	50	52.53	52.53	46.31	46.31	45.17	45.17	39.19	39.19	40.94	40.94
gpt-oss-120B (medium)	8192	66.22	66.22	69.28	69.28	48.99	48.99	46.74	46.74	40.09	40.09	43.62	43.62
gpt-oss-120B (high)	8192	59.64	82.45	61.43	82.05	44.74	47.62	44.72	48.77	35.14	39.8	38.7	42.51
	16384	68.51	76.46	71.03	78.04	45.86	46.17	47.42	48.28	38.74	40	41.61	42.47
Qwen/Qwen3-Coder-30B-A3B-Instruct	8192	28.63	28.63	27.4	27.4	21.92	21.92	24.49	24.49	26.8	26.8	24.61	24.61

Table 10: **CLUTRR Performance Test Splits:** We show the kinship relation type prediction accuracy (`Acc`) as well as non-null accuracy (`NNAcc`) for all the test splits within CLUTRR.

task performance. Finally, for fairness and consistency, we use sequence lengths comparable to those employed for CLUTRR and PBEBench-Lite, and we allow a single attempt per instance for each LLM.

F.4 Effect of More Examples

Table 21 illustrates the effect of varying the number of examples per PBE instance while keeping other factors, such as cascade distribution and relation type balance, constant using the PBEBench-Lite-MoreEg snapshot. The results for gpt-oss-20b and gpt-oss-120b show largely similar performance for gpt-oss-20b, but a decrease for gpt-oss-120b. This suggests that increasing the number of examples can sometimes make the task harder for LLMs, which is counterintuitive, as more examples would ideally simplify the task. Analysis of the average number of changes per program reveals that PBEBench-Lite has fewer absolute changes (1.56 words out of 5 on average) compared to PBEBench-Lite-MoreEg (7.07 words out of 50 on average). However, the relative number of changes is lower in PBEBench-Lite-MoreEg (14% vs. higher in PBEBench-Lite), indicating lower information density, which may explain why the task becomes harder despite having more examples.

F.5 Real SLI Results

We evaluate open-source LLMs on the real SLI dataset using `Pass@1`, `Edit_Sim`, and `Valid Rate`, with results reported in Table 13. Evaluation is performed on 21 PBE instances under substantially relaxed decoding constraints to estimate an upper bound on model performance. Specifically, we use a maximum sequence length four times larger than the corresponding PBEBench-Lite settings and a sampling budget of 32 attempts per instance. Models are additionally allowed to generate cascades of up to length 50, with each rewrite rule permitting α and β substrings in the `replace(`α`,` β`)` of up to five characters. Despite these concessions, all evaluated models struggle on real SLI. Only the gpt-oss family, and DeepSeek-R1-Distill-Qwen and Qwen3-32B with chain-of-thought prompting achieve `Pass@1` of 10% or higher. Among them, gpt-oss-120b performs the best, reaching a `Pass@1` of 33%.

F.6 Efficiency of Problem Proposer

For each desired ground-truth cascade length, our data generation procedure is designed to yield a balanced distribution across relation type categories, where each category is represented by a binary vector. Let U denote the ideal uniform distribution over all categories, and let Q denote the empirical distribution obtained from the generated data. To quantify the deviation of Q from the ideal U, we use the Kullback–Leibler (KL) divergence:

$$D_{\mathrm{KL}}(U||Q)=\sum_x U(x)\log\frac{U(x)}{Q(x)}$$

where x ranges over all relation type categories. By construction, $D_{\mathrm{KL}}(U||Q) \geq 0$ and equals zero only when $Q = U$. We also **apply smoothing** for categories where zero instances are observed in the empirical distribution (missing categories) to prevent the divergence from shooting up to infinity for these cases.

Model	Acc	PS	SS	Nulls
Codestral-22B	15.5	59.99	99.2	0
Qwen/Qwen2.5-32B-Instruct	27.3	72.1	99.8	0
Qwen/Qwen2.5-Coder-32B-Instruct	28.4	71.01	99.9	0
Qwen/QwQ-32B	43.5	44.66	45.6	0
Qwen/Qwen3-32B (with CoT)	44.4	46.49	47.8	0
Qwen/Qwen3-32B	32.4	71.8	99	0
Qwen/Qwen3-30B-A3B	41.7	49.77	61.1	0
DeepSeek-R1-Distill-Qwen-32B	43.2	59.9	85.6	0
gpt-oss-20B (low)	36	57.6	76.5	0
gpt-oss-20B (medium)	44.4	49.91	52.6	468
gpt-oss-20B (high)	40.2	40.68	41	589
gpt-oss-120B (low)	43.2	76.75	99.1	0
gpt-oss-120B (medium)	52.5	67.47	81.4	175
gpt-oss-120B (high)	47.2	47.29	47.3	524
Qwen/Qwen3-Coder-30B-A3B-Instruct	26.5	53.98	73.3	0

Table 11: **SLR-Bench Performance:** We compute the accuracy (fraction of cases successfully solved) (`Acc`), partial score (average number of examples correctly classified) (`PS`), `SS` (fraction of syntactically correct programs) and the number of null predictions (`Nulls`). We use the same max sequence length per model as CLUTRR.

Model	Basic			Easy			Medium			Hard		
	Acc	PS	SS	Acc	PS	SS	Acc	PS	SS	Acc	PS	SS
Codestral-22B	58.8	58.8	100	2.4	2.4	99.6	0.8	0.8	99.6	0	0	97.6
Qwen/Qwen2.5-32B-Instruct	90.8	90.8	100	15.6	15.6	100	1.2	1.2	99.6	1.6	1.6	99.6
Qwen/Qwen2.5-Coder-32B-Instruct	92.4	92.4	100	19.6	19.6	100	1.2	1.2	100	0.4	0.4	99.6
Qwen/QwQ-32B	96	96	96.8	62.8	62.8	63.2	9.6	9.6	12.8	5.6	5.6	9.6
Qwen/Qwen3-32B (with CoT)	94	94	94	69.2	69.2	74	11.2	11.2	14.4	3.2	3.2	8.8
Qwen/Qwen3-32B	92.8	92.8	100	35.2	35.2	100	1.2	1.2	97.2	0.4	0.4	98.8
Qwen/Qwen3-30B-A3B	99.6	99.6	100	61.6	61.6	80	5.2	5.2	27.6	0.4	0.4	36.8
DeepSeek-R1-Distill-Qwen-32B	99.2	99.2	100	63.6	63.6	88.8	8.8	8.8	70.4	1.2	1.2	83.2
gpt-oss-20B (low)	93.6	93.6	100	47.2	47.2	84.4	2.8	2.8	54	0.4	0.4	67.6
gpt-oss-20B (medium)	96.8	96.8	100	70	70	77.6	8.4	8.4	23.6	2.4	2.4	9.2
gpt-oss-20B (high)	96	96	98.4	59.2	59.2	60	5.2	5.2	5.2	0.4	0.4	0.4
gpt-oss-120B (low)	96.8	96.8	100	67.6	67.6	100	5.6	5.6	99.6	2.8	2.8	96.8
gpt-oss-120B (medium)	98	98	99.6	87.6	87.6	96.8	19.6	19.6	58.8	4.8	4.8	70.4
gpt-oss-120B (high)	98.8	98.8	98.8	75.2	75.2	75.6	12.4	12.4	12.4	2.4	2.4	2.4
Qwen/Qwen3-Coder-30B-A3B-Instruct	88.4	88.4	94	16	16	66.4	0.8	0.8	68.4	0.8	0.8	64.4

Table 12: **SLR-Bench Performance Test Splits:** We compute the accuracy (fraction of cases successfully solved) (`Acc`), partial score (average number of examples correctly classified) (`PS`), and `SS` (fraction of syntactically correct programs) for each split/curriculum tier of the test set.

Our data generation algorithm (the *problem proposer*) employs rejection sampling to enforce balance across categories. Initially, each sampled datapoint is accepted only if it maintains near-uniform coverage across categories. As sampling progresses, however, this constraint becomes increasingly difficult to satisfy, and the efficiency of rejection sampling deteriorates due to a growing rejection rate. To mitigate this, we introduce a *patience* parameter that limits the number of constrained steps. Once the patience threshold (e.g., 100000) is exhausted, the algorithm relaxes the balancing constraint and accepts datapoints unconditionally. This enables continued large-scale data generation, though at the cost of increased divergence $D_{\mathrm{KL}}(U||Q)$ from the uniform distribution. Importantly, this relaxation applies only to category balancing: the cascade length constraints remain strictly enforced, and each generated program must still achieve the target ground-truth cascade length

Model	Max Seq Len	First Code Block			Last Code Block		
		Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate
Codestral-22B	8192	0	31	81	0	31	81
Qwen2.5-32B-Instruct	8192	0	32	88	0	32	88
Qwen2.5Coder-32B-Instruct	8192	0	31	82	0	33	80
QwQ-32B	32768	5	63	94	5	63	94
Qwen/Qwen3-32B (with CoT)	32768	10	60	97	10	60	97
Qwen/Qwen3-32B	8192	5	60	79	5	60	79
Qwen3-30B-A3B	32768	5	9	97	5	9	97
DeepSeek-R1-Distill-Qwen-32B	32768	10	51	95	10	51	95
gpt-oss-20B (high)	32768	24	82	88	24	82	88
gpt-oss-120B (high)	32768	33	86	88	33	86	88
Qwen/Qwen3-Coder-30B-A3B-Instruct	8192	5	23	87	5	20	96

Table 13: **Real SLI Performance:** We evaluate all open source LLMs using the same metrics and prompts as PBEBench-Lite. Each LLM is given 32 attempts to solve the 21 PBE instances and each instance contains 50 input-output examples.

Dataset	r	p
CLUTRR	0.6364	0.035287
SLR-Bench	0.8091	0.002559
PBEBench-Lite	0.8273	0.001677

Table 14: **Benchmark Ranking Correlations:** Spearman rank correlations between open-source model rankings induced by inductive reasoning benchmarks and real SLI performance. For PBEBench-Lite and real SLI, rankings we use `Pass@1` as the primary metric with `Edit_Sim` as a tie breaker. For CLUTRR, Acc and NNAcc are used, while for SLR-Bench, Acc and Partial Score serve as the primary and tie-breaking metrics, respectively.

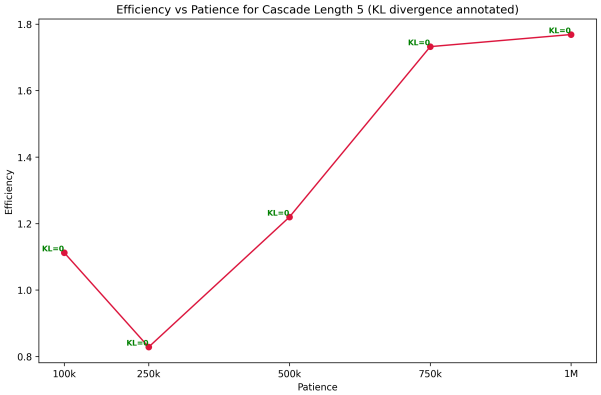


Figure 13: **Efficiency vs Patience:** Efficiency of the rejection sampling process for generating ground truth cascades of length 5 for various values of the patience parameter. The KL divergence from the ideal balanced distrbution is annotated per point with zero corresponding to achieving a perfectly balanced distribution.

and modify at least one example.

We visualize the efficiency of our data generation process for different values of the patience parameter τ—100,000; 250,000; 500,000; 750,000; and 1,000,000—across cascade lengths 5, 10, 15, 20, and 25, annotating each point with the KL divergence from the desired uniform distribution over all 16 relation type categories. Points with zero divergence (perfectly balanced distribution) are marked in green, while others are marked in red. The plots of efficiency versus patience for each cascade length are shown in Fig. 13–17. In all plots, the y-axis represents percentage efficiency (so 1 corresponds to 1% or 0.01). For cascade lengths 5 and 10, patience has little effect since perfect KL is always achievable; variations reflect random fluctuations across runs. For cascade lengths 15–25, perfect KL is no longer guaranteed and patience begins to influence efficiency. As expected, efficiency generally decreases with higher patience due to discarding more examples. For lengths 15 and 25, slightly lower KL can be achieved with greater patience, while for length 20, KL remains unchanged, indicating diminishing returns: higher patience does not necessarily reduce divergence from uniform distribution. These results motivate selecting a reasonable, but not excessive, patience value.

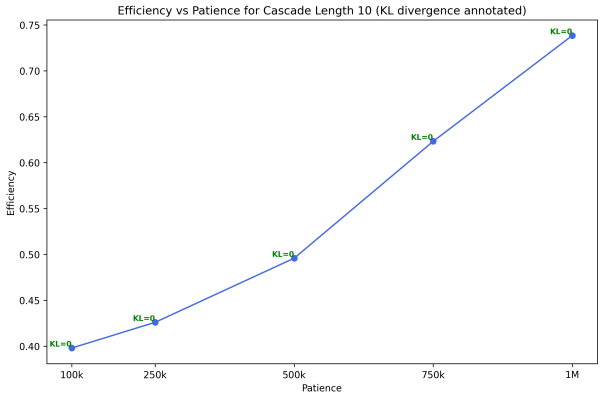


Figure 14: **Efficiency vs Patience:** Efficiency of the rejection sampling process for generating ground truth cascades of length 10 for various values of the patience parameter. The KL divergence from the ideal balanced distrbution is annotated per point with zero corresponding to achieving a perfectly balanced distribution.

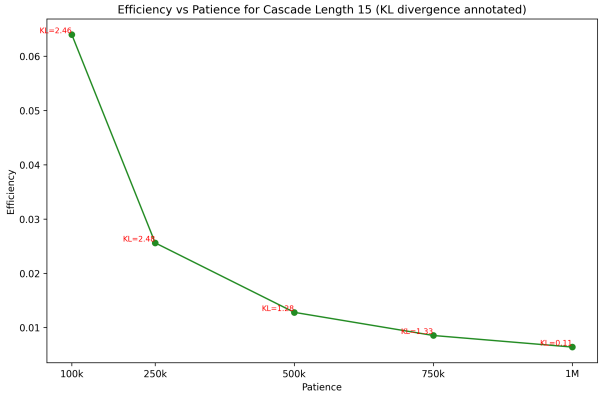


Figure 15: **Efficiency vs Patience:** Efficiency of the rejection sampling process for generating ground truth cascades of length 15 for various values of the patience parameter. The KL divergence from the ideal balanced distrbution is annotated per point with zero corresponding to achieving a perfectly balanced distribution.

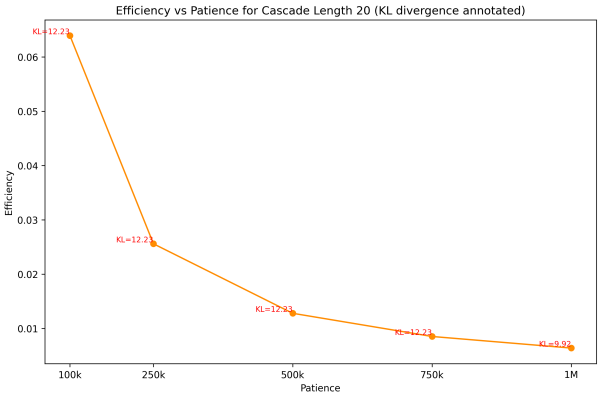


Figure 16: **Efficiency vs Patience:** Efficiency of the rejection sampling process for generating ground truth cascades of length 20 for various values of the patience parameter. The KL divergence from the ideal balanced distrbution is annotated per point with zero corresponding to achieving a perfectly balanced distribution.

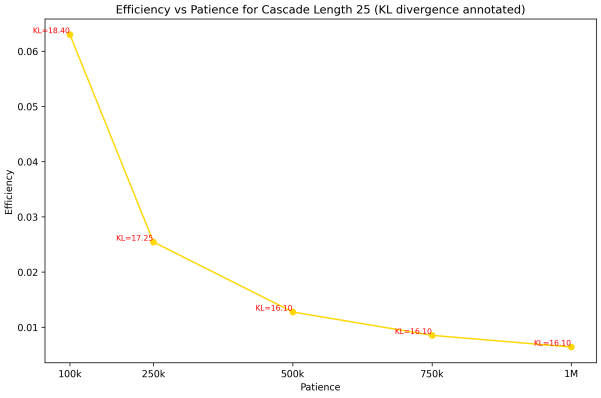


Figure 17: **Efficiency vs Patience:** Efficiency of the rejection sampling process for generating ground truth cascades of length 25 for various values of the patience parameter. The KL divergence from the ideal balanced distrbution is annotated per point with zero corresponding to achieving a perfectly balanced distribution.

Program Ordering Prompt

You are solving a **program ordering puzzle**. Given input-output string pairs and a scrambled list of string replacement programs, your goal is to determine the correct execution order.

```
## Background
```

Each program performs a Python string replacement: `replace("A", "B")` replaces all occurrences of `"A"` with `"B"`.

Why order matters:

- **Feeding:** One program creates substrings that another program can match.
  Example: `replace("a","bc")` followed by `replace("bc","x")`.
- **Bleeding:** One program removes substrings that another program would have matched.
  Example: `replace("ab","x")` followed by `replace("a","y")`.

```
## Your Task

**Inputs:** {inputs}

**Outputs:** {outputs}

**Scrambled Programs** (indices 0 to {n_minus_1}):
{programs_formatted}
```

Find the ordering `[i0, i1, ..., i_{n_minus_1}]` such that applying programs in that order transforms each input to its corresponding output.

```
## Approach
```

1. Trace through what each program does
2. Identify potential feeding/bleeding interactions
3. Reason about which programs must come before others
4. Verify your ordering produces the expected outputs

```
## Output Format
```

Provide your final answer as a JSON array of indices:

```
```json
[i0, i1, i2, ...]
```
```

Your ordering must be a permutation of `[0, 1, ..., {n_minus_1}]`.

F.7 Factorial Analysis

We conduct a factorial analysis to answer the following questions:

F.7.1 Effect of Long Chain of Thought Reasoning

We analyzed the effect of long chain-of-thought (LCoT) reasoning by selecting three models where it can be toggled on or off (Qwen3-32B, Claude-3.7-Sonnet, and Claude-4-Sonnet) on PBEBench-Lite (1008 instances) to evaluate its benefits. Independent variables included `model_id` (Qwen, Claude-3.7, or Claude-4), `reasoning` (LCoT enabled or not), `cascade_len` (ground-truth cascade length), and the presence of BFCC relations: `feeding`, `bleeding`, `counter-feeding`, and `counter-bleeding`. The dependent variable was binary Pass@1 (`passing`) per instance.

We fit a binary logistic regression model with `model_id`, `reasoning`, `feeding`, `bleeding`, `counter-feeding`, and `counter-bleeding` as nominal predictors, and `cascade_len` as a numeric covariate. All pairwise interaction terms were included. The Deviance goodness-of-fit test was non-significant, $\chi^2(6019) = 5398.5$, $p = 1.0$, indicating adequate model fit. The model explained 24.53% of the variance in passing (R^2_{adj}). Wald tests for main effects and significant interactions are summarized in Table 15.

F.7.2 Analysis of Models of Varying Capabilities

We analyzed what makes PBEBench-Lite instances difficult for three representative models of different capabilities. Independent variables included `cascade_len` (ground-truth cascade length) and the presence of BFCC relations: `feeding`, `bleeding`, `counter-feeding`, and `counter-bleeding`. The dependent variable was binary Pass@1 (`passing`). All models were fit with binary logistic regression, including all pairwise interaction terms. Results are summarized below.

Codestral-22B (weakest model). For Codestral-22B, only the effect of `cascade_len` was significant. Passing was less likely as the cascade length increased. Other predictors showed no detectable effects.

QwQ-32B (moderately good model). For QwQ-32B, the model fit the data adequately, $\chi^2(996) = 1946.72$, $p = .129$. The model explained 19.71% of variance (R^2_{adj}). Wald tests are summarized in Table 16. Cascade length, feeding, and bleeding all significantly reduced the probability of passing, while counter-feeding and counter-bleeding showed no effects. An interaction revealed that the joint presence of feeding and bleeding was less detrimental than either alone.

GPT-5 (strongest model). For GPT-5, the model fit the data adequately, $\chi^2(996) = 982.16$, $p = .617$. The model explained 17.28% of variance (R^2_{adj}). Wald tests are summarized in Table 18. Cascade length, feeding, and bleeding significantly reduced passing. Counter-feeding and counter-bleeding had no main effects, but an interaction showed that bleeding reduced passing only when counter-bleeding was absent.

F.8 Logistic Regression Analysis

We conduct a logistic regression analysis on gpt-oss-120b predictions on PBEBench for cascades of length 2 to 20, on the following factors influencing difficulty: ground-truth cascade length (`cascade_len`), the presence of BFCC relations: `feeding`, `bleeding`, `counter_feeding`, and `counter_bleeding`. The dependent variable was binary Pass@1 (passing) per instance. The logistic regression analysis reveals that all the analyzed factors have a statistically significant impact on the success or passing, with feeding, counter feeding, and cascade length having strong negative effects, with cascade length having the strongest impact. However, bleeding has a slight positive effect, and counter-bleeding has a very weak negative effect. This indicates that the presence of bleeding can make the problems easier to solve, while counter bleeding only has a small effect on increasing hardness.

F.9 Qualitative Analysis of Program Reordering Performance

We present a qualitative analysis of reasoning strategies employed by large language models on a program reordering task grounded in phonological rule ordering. Through examination of model responses across systems with visible chain-of-thought, we identify two distinct reasoning paradigms: (1) template-based constraint reasoning that explicitly models feeding/bleeding interactions between rules, and (2) exhaustive enumeration via brute-force permutation testing with forward simulation. Models employing constraint-based reasoning (e.g., Codestral-22B) correctly identify feeding relationships but often derive incorrect ordering

Term	df	χ^2	p	Interpretation
Total model	28	1027.68	.000	Deviance R^2 (adj) = 24.53%
cascade_len	1	415.35	.000	Passing is less likely as cascade_len increases
model_id	2	128.02	.000	The two sonnet models have a higher pass rate than Qwen
reasoning	1	18.70	.000	Reasoning models are more likely to pass
feeding	1	15.69	.000	Feeding reduces probability of passing
bleeding	1	36.10	.000	Bleeding reduces probability of passing
counter-feeding	1	21.37	.000	Counter-feeding reduces probability of passing
counter-bleeding	1	4.67	.031	Counter-bleeding increases probability of passing
model_id × reasoning	2	189.59	.000	For reasoning models, Qwen has a higher probability of passing than the two Sonnet models, which are not different from each other. For non-reasoning models, Claude 4 is better than Clause 3.7, which is better than Qwen.
feeding × bleeding	1	33.17	.000	Adding either feeding or bleeding reduces the probability of a pass, but both together is better than just feeding. Just bleeding is not worse than having both and not better than just feeding.
bleeding × counter-bleeding	1	30.86	.000	With counter-bleeding, bleeding loses its effect on probability of a pass.

Table 15: **Reasoning Manipulation Analysis on PBEBench-Lite:** Wald test results for main effects and significant interactions in the logistic regression predicting passing.

constraints—stating "Program A feeds Program B" but concluding A must precede B, the opposite of correct inference. Meanwhile, enumeration-based approaches (e.g., QwQ-32B, DeepSeek-R1) exhibit combinatorial collapse as problem size increases: response length grows from roughly 7,000 to about 28,000 characters while accuracy drops from nearly 90% to around 40% as the search space expands from 2 to 120 permutations. Notably, models rarely reference counterfeeding or counterbleeding despite these interactions being critical to the hardest instances. These findings reveal systematic gaps in how LLMs translate domain knowledge into valid constraint satisfaction and highlight the limitations of brute-force search for combinatorial reasoning tasks.

F.10 Confusion Matrices for Cascade Lengths

We analyze the length of model-predicted cascades against the ground truth cascades to analyze if the models tend to find more or fewer rules than the ground truth cascade for both successful/passing (Pass@1 = 1) and failure cases. The results on the PBEBench-Lite dataset across all models for successful cases are shown in Fig 19 and Fig. 20. The plots reveal that for successful cases, the models tend to largely find solutions of the correct length for shorter cascades, but for longer ground truth

Term	df	χ^2	p	Interpretation
Total model	11	197.43	.000	Deviance R^2 (adj) = 19.71%
cascade_len	1	114.54	.000	Passing is less likely as cascade_len increases
feeding	1	5.74	.017	Feeding reduces the probability of a pass
bleeding	1	7.37	.007	Bleeding reduces the probability of a pass
counter-feeding	1	.22	.637	No effect
counter-bleeding	1	.47	.495	No effect
feeding × bleeding	1	10.03	.002	Though bleeding or feeding alone reduce the probability of a pass, if they are both present, they do not.

Table 16: **QwQ-32B Analysis on PBEBench-Lite:** Wald test results for main effects and significant interactions in the logistic regression predicting passing.

Term	Coefficient	Interpretation
cascade_length	-0.521	Strongest predictor. Longer = harder.
counterfeeding	-0.291	Hardest relation.
counterbleeding	-0.068	Moderate difficulty.
bleeding	-0.034	Small negative effect.
feeding	-0.004	Negligible effect.

Table 17: **Difficulty Analysis of PBEBench-Lite-Perm:** Logistic regression coefficients for factors affecting the difficulty of program reordering and interpretation of the values.

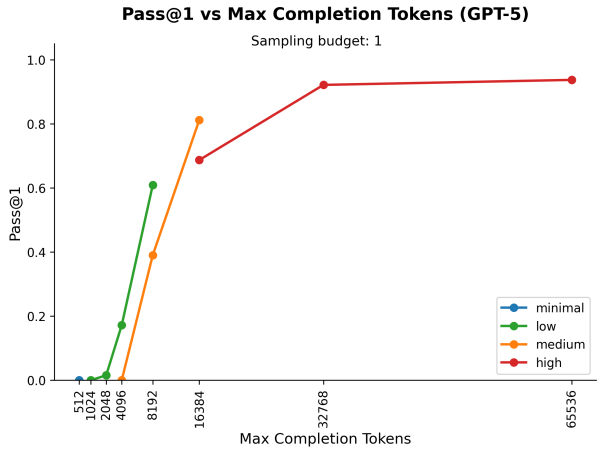


Figure 18: **Pass@1 vs Max Completion Tokens (sampling budget: 1):** Variation of Pass@1 vs Max Completion Tokens for various reasoning modes ranging from minimal, low, medium, to high.

cascades, they tend to find fewer solutions in general, but can surprisingly find shorter solutions as well. For failure scenarios, we note a bigger spread and almost see more cases of the LLMs generating longer programs than the ground truth. This might indicate that for the more complex cases, the LLMs tend to overthink and end up generating more complex cascades that don't work. We also see a large fraction of invalid programs, with this fraction growing more and more for longer ground truth cascades (more complex cases).

F.11 Confusion Matrices for BFCC Relations

We analyze the types of relations present in the model-generated program cascades and compare them against the ground truth relations and visualize the results via confusion matrices. The results are on the PBEBench-Lite dataset across all models and separated by whether the model succeeds or fails (based on Pass@1). We normalize each row (fraction of predicted cases for each possible relation type for a given ground truth type) and show the overall results for successful cases in Fig. 23 and failure cases in Fig. 24. While the per-model results for success cases and failure cases span from Fig 25 to Fig 30. We also plot a simplified version of the confusion matrix that looks at each relation at a time and analyze true positives, false positives, false negatives and true negatives spearately for each relation type for both passing (Fig. 21) and non-passing cases (Fig. 22) aggregated across all the models. These show an interesting pattern where for successful cases for almost all relation

Term	df	χ^2	p	Interpretation
Total model	11	158.16	.000	Deviance R^2 (adj) = 17.28%
cascade_len	1	63.76	.000	Passing is less likely as cascade_len increases
feeding	1	9.48	.002	Feeding reduces the probability of a pass
bleeding	1	4.95	.026	Bleeding reduces the probability of a pass
counter-feeding	1	1.16	.282	No effect
counter-bleeding	1	.54	.464	No effect
bleeding $\times$ counter-bleeding	1	8.50	.004	Bleeding only reduces probability of a pass if counter-bleeding is not present

Table 18: **GPT-5 Analysis on PBEBench-Lite:** Wald test results for main effects and significant interactions in the logistic regression predicting passing.

Term	Coefficient	p
intercept	2.152	0
feeding	-0.254	4.77e-19
bleeding	0.134	2.92e-6
counter_feeding	-0.176	6.27e-10
counter_bleeding	-0.08	0.005
cascade_len	-0.347	0

Table 19: **Logistic Regression Difficulty Analysis of gpt-oss-120b on PBEBench:** Model predictions were analyzed with a sampling budget of 32 and maximum sequence length of 16,384. Each attempt across the 1,216 instances was treated as a datapoint, yielding 38,912 datapoints in total.

types have relatively high false negative rates but it is especially bad for feeding (72% false negatives) and counter-feeding (62%) showing that even when the models can solve cases with ground truth cascades having these relation types they are highly biased against generating them. Interetingly we also see low false positives for the success/passing cases which is consistent with the tendencies of these models to try and not predict cascades that incorporate BFCC relations and the false positive rate is highest for feeding the hardest relation type. Finally for failure cases we note that there is a high false negative rate for every relation type ranging between 75% to 80%, consistent with the fact that the evaluated models try to avoid predicting BFCC relations and perhaps for cases where they are needed to find a correct solution, they fail to find one.

Cascade Length	Pass@1	
	First Code Block	Last Code Block
2	1	1
3	1	1
4	0.9688	0.9688
5	1	1
6	0.9688	0.9688
7	0.9688	0.9688
8	0.9688	0.9688
9	0.8438	0.8438
10	0.8438	0.8438
11	0.875	0.875
12	0.7656	0.7656
13	0.6719	0.6719
14	0.4531	0.4531
15	0.5	0.5
16	0.3906	0.3906
17	0.2812	0.2812
18	0.1719	0.1719
19	0.0781	0.0781
20	0.0469	0.0469

Table 20: **Performance across Cascade Lengths (gpt-oss-120b):** Variation of the performance of gpt-oss-120b across cascades of length 2-20 with sampling budget of 32 and max sequence length of 16384.

G Use of AI Assistants

We made limited use of AI assistants during the preparation of this manuscript, primarily for rephrasing, paraphrasing, and performing grammatical and stylistic checks. All technical content,

Model	First Code Block			Last Code Block		
	Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate
gpt-oss-20b ★■	0.3958	0.4662	0.9748	0.3958	0.4662	0.9748
gpt-oss-120b ★■	0.5542	0.7018	0.9211	0.5542	0.7018	0.9211

Table 21: **PBEBench-Lite-MoreEg Performance:** We compute the Pass@1 and edit similarity as the coarse and fine-grained evaluation, respectively, for each model. ■- indicates mixture-of-experts (or MoE) model and ★- indicates reasoning on for model.

Cascade Length	Sampling Budget	Pass@1
20	1	0.1543
	2	0.2416
	3	0.2974
	4	0.3377
25	1	0.0547
	2	0.0911
	3	0.1172
	4	0.1406
30	1	0.0117
	2	0.0234
	3	0.0351
	4	0.0469

Table 22: **Performance across Cascade Lengths (GPT-5):** Variation of the performance of GPT-5 across cascades with sampling budget ranging from 1 to 4, medium reasoning effort and 65536 max completion tokens (which includes both reasoning and output tokens).

analysis, and conclusions are solely those of the authors.

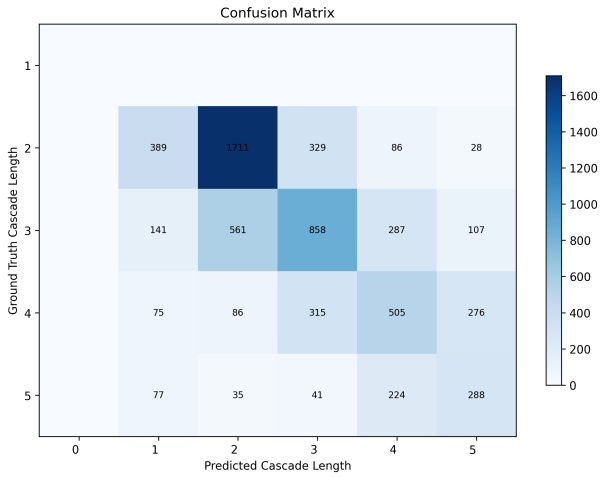


Figure 19: **Cascade Length Confusion Matrix for Success (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of cascade lengths in the model prediction vs ground truth. 0 length on the predicted side corresponds to cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

Sample Budget	First Code Block	Last Code Block
1	0.1543	0.1543
2	0.2416	0.2416
3	0.2974	0.2974
4	0.3377	0.3377
5	0.3694	0.3694
6	0.3956	0.3956
7	0.418	0.418
8	0.4375	0.4375

Table 23: **Performance across sampling budget (GPT-5):** Variation of the performance of GPT-5 with sampling budget for cascade length of 20 for medium reasoning effort and 65536 max completion tokens.

Max Completion Tokens	Reasoning Effort	Avg Tokens Used	Pass@1	Null Responses (%)	Non null Correct Responses (%)	Non null Incorrect Responses (%)
512	minimal	194	0	0	0	100
1024	low	1024	0	100	0	0
2048	low	2046	0.0156	98.4	1.56	0
4096	low	3909	0.1719	72	17.19	10.81
4096	medium	4096	0	100	0	0
8192	low	5041	0.6094	3.1	60.94	35.96
8192	medium	7726	0.3906	61	39	0
16384	medium	9738	0.8125	4.7	81.25	14.05
16384	high	13358	0.6875	23	68.75	8.2
32768	high	14045	0.9219	0	92.19	7.81
65536	high	14549	0.9375	0	93.75	6.25

Table 24: **Performance across reasoning efforts and max completion tokens (GPT-5):** Variation of the performance of GPT-5 with reasoning effort and max completion tokens for cascade length of 10, and sampling budget of 1.

Sampling Budget	First Code Block			Last Code Block			Nulls
	Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate	
1	0.25	0.7084	0.9776	0.25	0.7084	0.9776	1
2	0.5	0.7931	0.9752	0.5	0.7931	0.9752	1
4	0.6094	0.9081	0.983	0.6094	0.9081	0.983	5
8	0.7344	0.9477	0.9588	0.7344	0.9477	0.9588	4
12	0.7812	0.9619	0.9684	0.7812	0.9619	0.9684	10
16	0.8281	0.9679	0.9706	0.8281	0.9679	0.9706	11
20	0.8594	0.9691	0.9788	0.8594	0.9691	0.9788	26
24	0.7969	0.9725	0.9638	0.7969	0.9725	0.9638	3
28	0.8438	0.9644	0.9797	0.8438	0.9644	0.9797	12
32	0.8438	0.9739	0.9806	0.8438	0.9739	0.9806	20
64	0.9062	0.9808	0.9855	0.9062	0.9808	0.9855	27

Table 25: **Performance across sampling budget (gpt-oss-120b):** gpt-oss-120b performance with sampling budget for max sequence length of 16384 on the 64 PBEBench instances with ground-truth cascade length 10. The nulls column counts cases (out of $Sampling\ Budget \times 32$) where the chain of thought fails to terminate within the max sequence length.

Sampling Budget	First Code Block			Last Code Block			Nulls
	Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate	
32	0.8438	0.9739	0.9806	0.8438	0.9739	0.9806	20
32	0.8594	0.9716	0.9814	0.8594	0.9716	0.9814	16
32	0.8281	0.9772	0.9745	0.8281	0.9772	0.9745	32

Table 26: **Performance variance per run (gpt-oss-120b):** We analyze the variance exhibited by gpt-oss-120b across all the metrics for a given run. We conduct this experiment on the 64 instances corresponding to cascade length 10 which is also used for all the scaling experiments and use a sampling budget of 32 and max sequence length of 16384, the same parameters used for evaluation of gpt-oss-120b on PBEBench. The nulls column counts cases (out of $Sampling\ Budget \times 32$) where the chain of thought fails to terminate within the max sequence length.

Max Seq Length	First Code Block			Last Code Block			Nulls
	Pass@1	Edit Sim	Valid Rate	Pass@1	Edit Sim	Valid Rate	
2048	0	0.0141	1	0	0.0141	1	2043
3840	0.2812	0.5479	0.9717	0.2727	0.4099	1	1799
5632	0.625	0.8447	0.9724	0.625	0.8447	0.9724	1212
7424	0.7656	0.9473	0.9705	0.7656	0.9473	0.9705	607
9216	0.8594	0.9631	0.9875	0.8594	0.9631	0.9875	305
12800	0.875	0.9706	0.9788	0.875	0.9706	0.9788	80
14592	0.875	0.9748	0.9825	0.875	0.9748	0.9825	28
16384	0.8438	0.9739	0.9806	0.8438	0.9739	0.9806	20
32768	0.9219	0.9819	0.9838	0.9219	0.9819	0.9838	0
65536	0.9062	0.9755	0.9838	0.9062	0.9755	0.9838	0

Table 27: **Performance across max sequence length (gpt-oss-120b):** gpt-oss-120b performance with max sequence length at sampling budget 32 on the 64 PBEBench instances with ground-truth cascade length 10. The null column counts cases (out of $64 \times 32 = 2048$) where the chain of thought fails to terminate within the sequence limit.

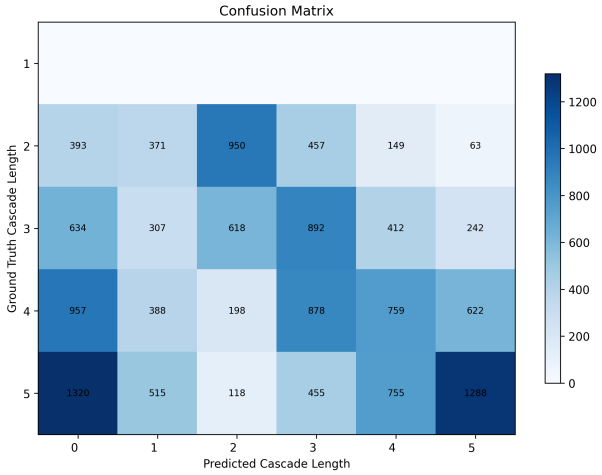


Figure 20: **Cascade Length Confusion Matrix for Failure (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of cascade lengths in the model prediction vs ground truth. 0 length on the predicted side corresponds to cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

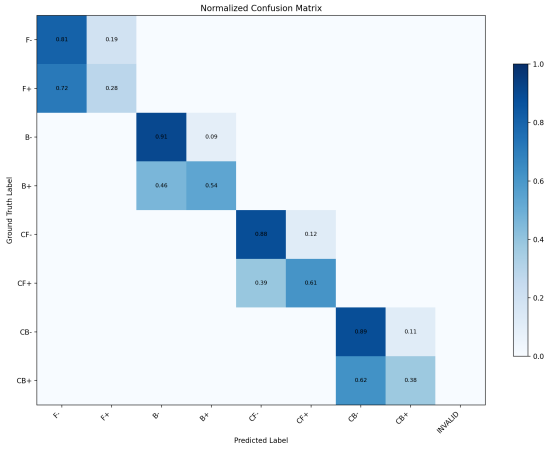


Figure 21: **Simplified Relation Type Confusion Matrix for Success (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and success denotes Pass@1 = 1 with a sampling budget of 1.

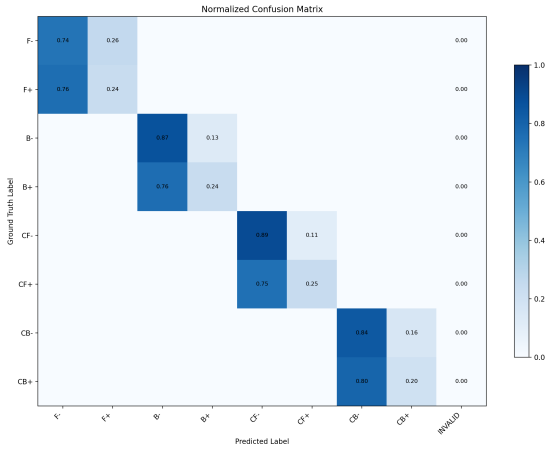


Figure 22: **Simplified Relation Type Confusion Matrix for Failure (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

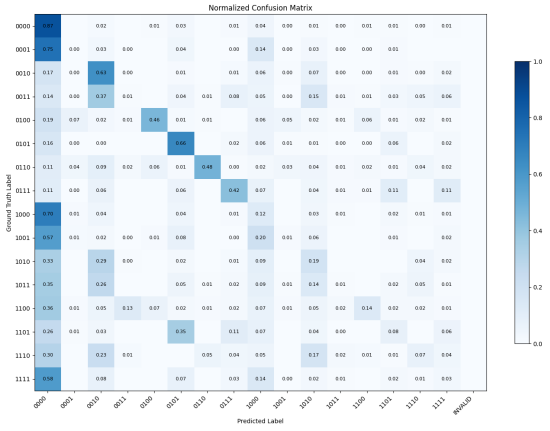


Figure 23: **Relation Type Confusion Matrix for Success (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and success denotes Pass@1 = 1 with a sampling budget of 1.

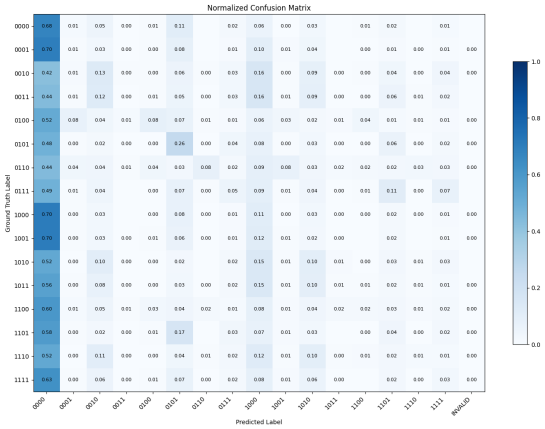


Figure 24: **Relation Type Confusion Matrix for Failure (All Models PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are averaged across all models on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

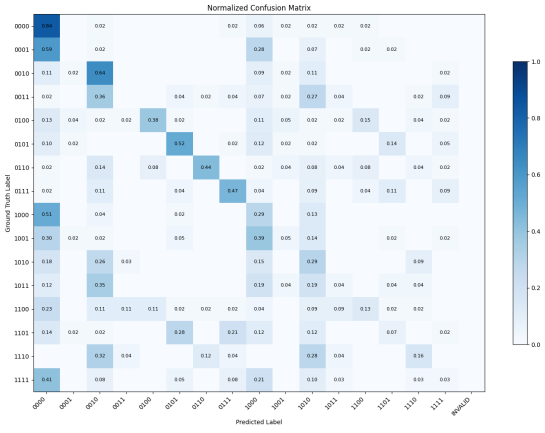


Figure 25: **Relation Type Confusion Matrix for Success (GPT-5 PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for GPT-5 on PBEBench-Lite, and success denotes Pass@1 = 1 with a sampling budget of 1.

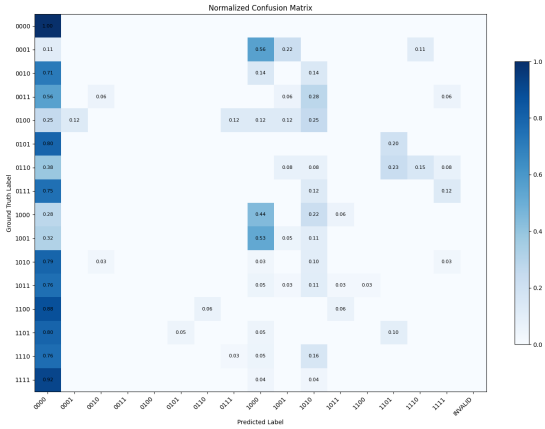


Figure 26: **Relation Type Confusion Matrix for Failure (GPT-5 PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for GPT-5 on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

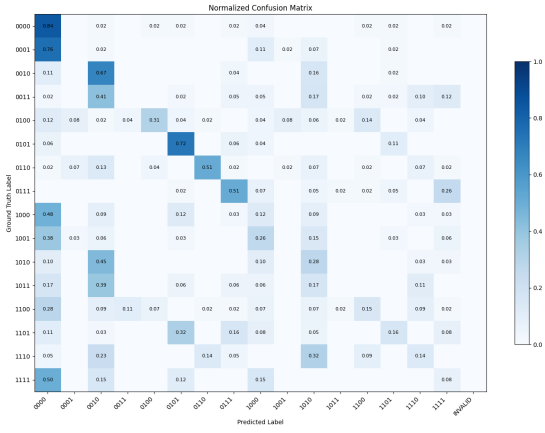


Figure 27: **Relation Type Confusion Matrix for Success (gpt-oss-120b PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for gpt-oss-120b on PBEBench-Lite, and success denotes Pass@1 = 1 with a sampling budget of 1.

Figure 28: **Relation Type Confusion Matrix for Failure (gpt-oss-120b PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for gpt-oss-120b on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

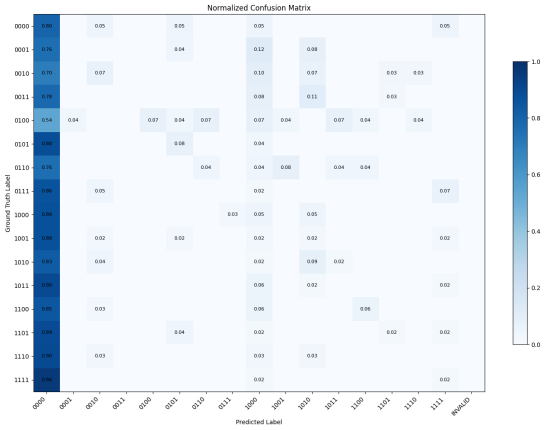


Figure 29: **Relation Type Confusion Matrix for Failure (gpt-oss-120b PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for gpt-oss-120b on PBEBench-Lite, and failure denotes Pass@1 = 0 with a sampling budget of 1.

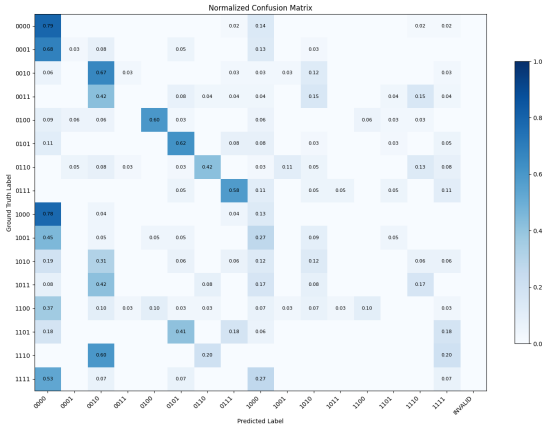


Figure 30: **Relation Type Confusion Matrix for Success (gpt-oss-120b PBEBench-Lite)**: Confusion matrix showing the distribution of relation types in the model prediction vs ground truth. INVALID category indicates cases where the model fails to generate a valid cascade at all. The results are for gpt-oss-120b on PBEBench-Lite, and success denotes Pass@1 = 1 with a sampling budget of 1.