CauSim: Scaling Causal Reasoning with Increasingly Complex Causal Simulators

Nicolás Astorga[1]* **Anita Kriz**[1]* **Mihaela van der Schaar**[1]
[1]DAMTP, University of Cambridge, Cambridge, UK
nja46@cam.ac.uk, ak2680@cam.ac.uk

Abstract

Despite surpassing human performance across mathematics, coding, and other knowledge intensive tasks, large language models (LLMs) continue to struggle to causally reason. One obstacle is the target data itself: causal systems are complex and often expressed in non-executable forms, and ground-truth answers to causal queries are inherently scarce. We introduce **CauSim**, a framework that turns causal reasoning from a scarce-label problem into a scalable, supervised one. It constructs *increasingly complex causal simulators*: executable structural causal models (SCMs), incrementally built by LLMs, that scale to globally complex systems while maintaining verifiable answers to any causal query. **CauSim** operates across representations by formalizing non-executable causal knowledge into code, allowing for data augmentation, and informalizing executable SCMs into natural language, enabling supervision in previously unsupervisable representations. We structure our research into two parts: (1) how to construct increasingly complex causal simulators, and (2) a systematic study of what **CauSim** enables, demonstrating generalization across representations, consistent gains from curriculum scaling and data volume, LLM self-improvement through self-generated simulators, and data augmentation via formalization of existing domain knowledge. Code: GitHub Repository.

1 Introduction

Understanding how a system behaves and reasoning causally about that system are two distinct capabilities. Knowing the laws governing a physical system, for example, does not by itself tell us where a ball would have landed had it been pushed with a different force. Even when the causal mechanisms are explicitly provided, answering such what-if questions requires using the observed world to infer the relevant conditions and reasoning through how a hypothetical change would propagate through the system. Despite surpassing human performance across mathematics [1], coding [2, 3], and other knowledge-intensive tasks [4, 5], Large Language Models (LLMs) struggle to causally reason [6, 7, 8, 9].

Why is causal reasoning hard, even when the rules are known? Consider the counterfactual query in Fig. 1: *How big would Tumor X have been if Drug Y had been given?* Suppose we are already given the relevant causal mechanisms: how Drug Y affects Blocker Z, and how Blocker Z influences Tumor X. Reasoning about the query is still hard for two reasons. First, ▶ **[C1] scale**: as the number of variables and interactions grows, correctly composing mechanisms and tracing their effects becomes increasingly challenging. Second, ▶ **[C2] representation**: even when the causal knowledge is available, it may be expressed in non-executable forms (e.g., natural language), requiring the underlying mechanisms to be correctly extracted and composed before the query can be answered. In principle, LLMs could be trained to overcome both challenges. However, ▶ **[C3] supervision**: ground-truth interventional outcomes are limited and costly, while individual counterfactual outcomes

*Equal contribution.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

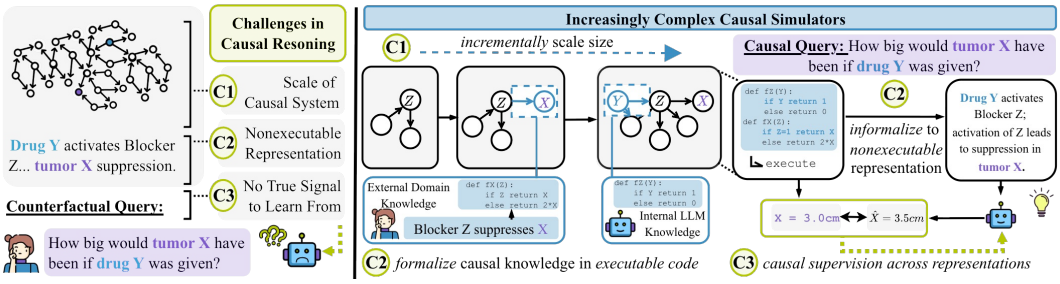


Figure 1: **Overview of CauSim. (Left)** Causal reasoning faces three challenges: **[C1]** scale, **[C2]** non-executable representations, and **[C3]** lack of ground-truth supervision. **(Right) CauSim** addresses these through increasingly complex causal simulators: SCMs are built *incrementally* to scale complexity **[C1]**; non-executable causal knowledge is *formalized* into executable code and executable SCMs are *informalized* into natural language **[C2]**; enabling causal supervision **[C3]**

are unobservable [10, 11, 12]. This makes scalable direct supervision of causal reasoning difficult, distinguishing it from other modes of reasoning.

Key questions. How can we get supervision for causal reasoning across scales and representations?

Increasingly Complex Causal Simulators. Inspired by recent work leveraging the executability of code for scalable supervision [13], we argue that causal reasoning needs its own scalable, verifiable training environment. We introduce *increasingly complex causal simulators*: structural causal models (SCMs)[1], constructed by LLMs, that are expressible across representations. Our insight is two-fold. First, by formalizing SCMs as code, any causal query can be answered via code execution. Second, by leveraging LLMs' code generation and contextual capabilities, we can build causal simulators *incrementally*, with each modification compilable to ensure causal consistency. This enables complex simulators that would be difficult to construct in a single step. To operate across causal representations, we utilize LLMs to (1) informalize executable representations into non-executable forms, facilitating supervised training in previously unlabeled distributions, and (2) formalize non-executable representations into executable form, enabling data augmentation with both interventional and counterfactual samples. In all, **CauSim** turns causal reasoning from a complex scarce-label problem into a scalable, supervised one.

Key Findings. We evaluate **CauSim** by building and training on causal simulators over meaningless variables to reduce confounding from prior domain knowledge. We first demonstrate that step-by-step construction scales more reliably than one-shot generation, enabling the construction and translation of larger SCMs (Sec. 4). We then study what these causal simulators enable for training (Sec. 5). **(1) Generalization:** Training on executable SCMs improves causal reasoning on unseen medical content and non-executable representations, with further gains from additional training on non-executable SCMs over meaningless variables; improvements also transfer to independently constructed external benchmarks. **(2) Scaling:** Progressive complexity curricula and larger amounts of simulator-generated supervision consistently improve performance, particularly on more complex SCMs. **(3) Self-improvement:** An LLM can improve its causal reasoning using SCMs that it generates itself, demonstrating one-step bootstrapping from self-generated environments. **(4) Data augmentation:** Formalizing existing causal knowledge as executable SCMs enables additional supervision that improves performance in the original domain. **(5) Anchoring performance:** Models trained under **CauSim** outperform larger untrained models from the same family.

Contributions. Our contributions are two-fold: (1) We propose **CauSim**, a novel framework for constructing increasingly complex causal simulators across scales and representations (Sec. 4). (2) We systematically study what **CauSim** enables, demonstrating ▶ generalization to out-of-distribution and non-executable representations, ▶ consistent gains by curriculum scaling and data volume, ▶ LLM self-improvement, and ▶ data augmentation (Sec. 5).

2 Related work

Evaluating Causal Reasoning. There is an abundant literature on the *evaluation* of LLM causal reasoning, including on textual problems derived from SCMs [6, 16, 8], code and mathematical

[1] Structural causal models, introduced by Judea Pearl, are a widely used formalism for representing causal systems in the causal inference literature. We provide a brief overview in Sec. 3.1. See [14, 15, 12] for more details.

Table 1: Comparison of **CauSim** with related work across key properties. ✓ indicates supported; ✗ indicates not supported.

Improve Reasoning | Evaluate Causal Reasoning | Improve Causal Reasoning

Method	Causal	Scalable	Executable	Training	Formalize	Informalize
Absolute Zero [13]	✗	✓	✓	✓	✗	✗
CLadder [6]	✓	✗	✗	✗	✗	✗
Re-Imagine [24]	✓	✗	✓	✗	✓	✓
Hüyük et al. [34]	✓	✗	✗	✓	✗	✗
Exec. Counterfactuals [35]	✓	✗	✓	✓	✗	✗
CauSim. (Ours)	✓	✓	✓	✓	✓	✓

domains [17, 18], and abstract out-of-distribution regimes [19]. Complementary studies consider specific causal reasoning properties: compositional causal inference [20], probabilistic necessity and sufficiency [21], and counterfactual reasoning in hypothetical [22] and multimodal settings [23]. *All of these methods rely on static, human-curated datasets.* Re-Imagine [24] is a notable exception that applies mutations to create causal queries from benchmark-derived causal graphs, but they do *not* study scaling graph complexity. In contrast to all of the aforementioned methods, our work is not concerned with evaluation. We focus on improving LLM causal reasoning.

Improving Causal Reasoning. Although there is substantial research on improving *general* reasoning [25, 26, 27, 28, 29, 30, 31, 32], improving *causal* reasoning remains limited. Liu et al. [33] demonstrate that prompting LLMs with code significantly improves causal reasoning performance. Hüyük et al. [34] finetune causal reasoning via SFT and DPO on datasets of factual and counterfactual examples over binary variables. More recently, executable code-based SCMs have been explored as a setting for causal supervision [35], though restricted to *fixed*, template-generated graphs. In contrast, our framework exploits executability to incrementally scale causal simulators, enabling globally complex systems with verifiable supervision.

Improving Reasoning through LLM-Generated Data. A body of work explores using LLMs to generate training data and environments to improve reasoning performance [36, 37, 13]. In particular, Absolute Zero [13] proposes and solves tasks to maximize learning progress. We build on this scalable, environment-centric body of work, but specifically target causal reasoning by having LLMs generate increasingly complex causal simulators as verifiable environments. To our knowledge, this is the first work to apply LLM-generated environments to improve causal reasoning.

3 Formalism and Background

Our framework is designed to build executable supervision across *scales* and *representations*. For clarity, we first formalize the problem of answering causal queries (§3.1) and outline the key challenges this formalization reveals (§3.2).

3.1 Problem Setting

We consider the task of answering a causal query q about an underlying data generating process (DGP). We adopt Pearl's SCM framework as the underlying DGP.

Definition 3.1 (Structural Causal Model (SCM) [14])**.** A (SCM) is a tuple: $\mathcal{M} \coloneqq \langle \mathbf{U}, p(\mathbf{u}), \mathbf{V}, \mathbf{F} \rangle$, where $\mathbf{U} = \{U_i\}_{i=1}^m$ is a set of exogenous variables distributed according to $p(\mathbf{u})$, $\mathbf{V} = \{V_i\}_{i=1}^n$ is a set of endogenous variables, and $\mathbf{F} = \{f_i\}_{i=1}^n$ is a collection of structural assignments such that each V_i is generated as: $v_i = f_i(\mathrm{pa}_i, u_i)$, where $\mathrm{pa}_i \subseteq \mathbf{V} \setminus \{V_i\}$ denotes the endogenous parents of V_i, and u_i denotes the corresponding exogenous context.

As in [35], the SCM is provided as input to the LLM. In our work, we make the insight that the same SCM $\mathcal{M}$ may be expressed in substantially different representations $R \in \mathcal{R}$, for instance as natural language descriptions or domain-specific diagrams. We define a representation $R(\mathcal{M})$ as a complete encoding of the SCM $\mathcal{M}$. A representation is *executable* if $R(\mathcal{M})$ can be directly compiled to yield answers to any causal query; otherwise it is *non-executable*. Our framework operates over both, with translation between them defined in §4. Our primary interest is in how reasoning difficulty scales with increasing n and varying representations $\mathcal{R}$.

Types of Causal Queries. We consider four types of causal queries over SCMs. At the core are two primitive processes: *Deduction*: predicting endogenous variables $\mathbf{V}$ from exogenous variables $\mathbf{U}$ via forward propagation through $\mathbf{F}$, and *Abduction*: inferring exogenous values $\mathbf{U}$ from observed

endogenous values $\mathbf{V}$. Building on these, *Intervention* is a modified form of deduction in which one or more endogenous variables are externally set via the $\mathrm{do}(\cdot)$ operator. *Counterfactual* queries combine both: abduction first infers exogenous variables consistent with observed outcomes, which are then propagated forward through the intervened SCM. Formal definitions in Appendix Q.

The Goal. Let $\mathbf{v}$ denote an observation of endogenous variables $\mathbf{V}$, and q a causal query over $\mathcal{M}$. Our goal is to train an LLM π_θ to answer q over SCMs of varying scale n and representations $R \in \mathcal{R}$ given $R(\mathcal{M})$, by minimizing a loss between its prediction $\hat{a} = \pi_\theta(R(\mathcal{M}), q, \mathbf{v})$ and the ground-truth answer a: $\min_\theta \ \mathbb{E}[\mathcal{L}(\hat{a}, a)]$

To update the LLM parameters θ, we use reward-based bootstrapping, with the reward determined by the correctness of the predicted answer $\hat{a}$. Other forms of supervision could also be used.

3.2 Challenges.

The above goal formulation identifies three aspects that make causal reasoning difficult:

[C1] System Scale, n. As the size of the SCM increases, answering causal queries becomes increasingly difficult. Consider the causal system in Fig. 1, where Drug Y activates Blocker Z across a long network pathway to suppress Tumor X. To understand the effect of an intervention on Y, one must trace how Y influences Z, then determine how activation of Z affects X. Even though all interactions are fully specified, reasoning through these sequential and interacting steps is nontrivial.

[C2] Representation, R. Some representations may be much easier to interpret than others. For example, a causal mechanism written as a formal logical program may be easier to interpret than the same mechanism described in a lengthy paragraph. Effective causal reasoning within any representation requires identifying the actual mechanisms from superfluous content, composing them correctly, and reasoning over the resulting complete structure.

[C3] Limited Supervision, a. In principle, challenges **[C1-2]** could be addressed with sufficient ground-truth data using training techniques. The problem is that this ground-truth data does not exist for causal reasoning. Interventional data is limited and costly to obtain, and counterfactual outcomes are inherently unobservable. As a result, the ground-truth answers required to directly supervise the causal queries described in Section 3.1 may not (and likely do not) exist in real world settings.

The Core Challenge. Together, these challenges point to available data as a limitation in improving causal reasoning. LLMs may struggle to reason over large systems and varying representations, and the ground-truth supervision that could be used to train them is scarce.

4 The Method: Building Increasingly Complex Causal Simulators

The limitations of available data motivate our research question: *if we cannot use existing data to train LLMs across the full space of causal queries, can we simulate the data we need?*

Overview. Our method, **CauSim**, uses LLMs to construct increasingly large SCMs as executable Python programs (Sec. 4.1). Since real-world causal knowledge is usually not expressed in executable form, our framework: (1) translates executable SCMs into non-executable representations, which provides verifiable supervision in previously unsupervisable settings, and (2) translates non-executable representations into executable SCMs, enabling existing causal knowledge to be incorporated into the data-generation process (Sec. 4.2). This produces supervised data across scales **[C1]** and representations **[C2]**, with ground-truth supervision across all causal query types **[C3]** (Sec. 4.3).

Motivated by recent work on LLM-generated training environments (Sec. 2), our framework uses LLM-based SCM construction to jointly support scalable simulator growth and translation across causal representations. Further discussion of this design choice is provided in H.

4.1 Scaling Structural Causal Models

As n increases, it becomes increasingly difficult for LLMs to constuct SCMs that maintain fidelity to causal assumptions and preserve code executability (see Experiment 1 below). We address this by constructing executable SCMs *incrementally* through a four-stage procedure: ▶ **specification**, ▶ **planning**, ▶ **execution**, and ▶ **verification**. In the interest of space, we provide a visual overview in Fig. 2 and defer the full details and walk-through example to Appendix A.1.

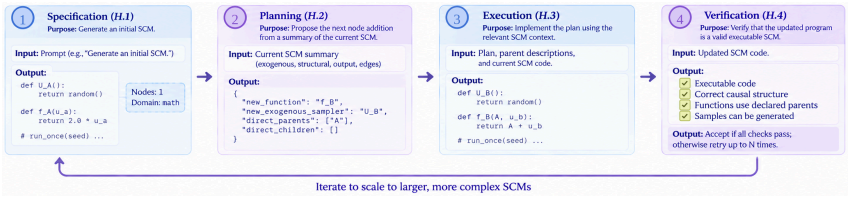


Figure 2: **CauSim (executable component)** uses LLMs to generate SCMs incrementally, node-by-node, starting from a small initial SCM. At each stage, context is restricted: planning uses only a high-level semantic summary of the current SCM, while execution receives only the code for affected child nodes and descriptions of relevant parents. Planning, execution, and verification repeat until the target size is reached.

To validate the choice of incremental construction of SCMs, we compare incremental SCM generation versus one-shot generation at increasing scales in EXPERIMENT 1.

EXPERIMENT 1. DOES INCREMENTAL SCM GENERATION ENABLE SCALABILITY?

Method. We compare two methods for constructing an SCM with n structural mechanisms and n corresponding exogenous samplers as executable *code*: (i) *One-shot generation*: the LLM is prompted to output the full SCM in a single specification step and (ii) *Incremental generation*: the LLM first specifies a 1-node SCM, and then incrementally grows by adding a single node and its corresponding exogenous node at a time using our planning - execution loop.

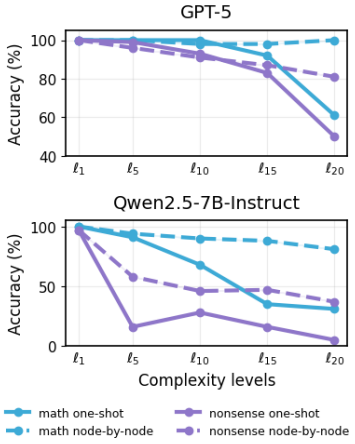


Figure 3: **Node-by-node generation improves SCM generation *at scale*.**

Experimental Setup. We evaluate these two approaches on two domains represented in executable format: a symbolic MATH domain and a NONSENSE domain, consisting of variables of 4-letter nonsensical strings (a design choice we discuss in Sec. 5). The full prompts used to generate SCMs in each domain are provided in Appendix N and examples of MATH and NONSENSE SCMs are shown in Appendices A.1.1 and A.7.1. To study if generating increasingly large SCMs is challenging across models of different capacities, we test both GPT-5 and QWEN-2.5-7B-INSTRUCT. Because QWEN-2.5-7B-INSTRUCT is less consistent in following instructions, we allow up to three attempts per model call to meet verification criteria, whereas GPT-5 is allowed a single attempt. Accuracy is measured as the fraction of 100 trials that pass all verification checks.

Results. Figure 3 compares one-shot and incremental SCM generation performance as a function of SCM size. Across domains and LLMs, one-shot generation degrades more rapidly as n increases than our node-by-node approach.

✓ **Takeaway.** Incremental SCM construction scales more reliably than one-shot generation, enabling the generation of increasingly large SCMs while preserving structural validity.

4.2 Translating Structural Causal Models

In many settings, causal queries are expressed in *non-executable* representations—for example, natural-language descriptions that cannot be directly compiled or executed **[C2]**. To operate in these settings, **CauSim** uses two complementary translation steps. First, *informalization* translates an executable SCM into a semantically equivalent non-executable representation, approximating those encountered in practice while retaining an executable counterpart that provides ground-truth answers to causal queries. Second, *formalization*, analogous to optimization model autoformulation [38], translates causal knowledge from a non-executable representation into an executable SCM, allowing this knowledge to initialize a simulator that can then be augmented using our incremental construction procedure. Together, these steps enable a *formalization* $\rightarrow$ *augmentation* $\rightarrow$ *informalization* pipeline for generating verifiable supervision from existing causal knowledge.[2] Further implementation details and prompts are provided in Appendix O.

Summary. Building on our four-stage incremental approach, we formalize non-executable causal knowledge *step by step*, restricting each planning step to parent and child relationships explicitly

[2]Formalization is distinct from causal reasoning: it constructs an initial executable SCM from a source representation, which can subsequently be modified or expanded using the incremental construction procedure in Sec. 4.1.

supported by the source representation. Conversely, we informalize executable SCMs into non-executable representations. As a proxy for verifying semantic preservation, we formalize the resulting representation back into an executable SCM and confirm that its semantic context and execution remain consistent. We evaluate step-by-step formalization against a one-shot approach below.

EXPERIMENT 2: CAN NON-EXECUTABLE KNOWLEDGE BECOME EXECUTABLE SCMS?

Method. We evaluate our formalization procedure across different source representations. The NIH Stroke Scale (NIHSS) is a natural-language, point-based clinical scoring system for stroke severity, where individual items measure neurological functions. The Bayesian networks ASIA and CANCER from `bnlearn` [39] use conditional probability tables to specify the distribution of each variable given its parents. Examples of the source representations and corresponding executable SCMs are provided in Appendices A.3.4 and A.3.6.

Experimental Setup. We use GPT-5 to generate a reference SCM from these dataset and manually verify its correctness against the original representation. This reference SCM is used to measure the accuracy of the generated SCMs. We then prompt QWEN-2.5-7B-INSTRUCT to formalize the SCM either in (i) one-shot or (ii) incrementally using our planning-execution loop. Accuracy is measured as the fraction of 100 trials that pass all verification checks and match the causal structure of the GPT-5 generated SCM. All prompts are shown in Appendix A.3.

Dataset	Step-by-step	One-shot
NIHSS	**52/100**	20/100
ASIA	**84/100**	69/100
CANCER	**74/100**	51/100

Table 2: **Node-by-node outperforms one-shot formalization.**

Results. Table 2 reports the accuracy of one-shot versus incremental SCM construction from the NIHSS. Node-by-node generation significantly outperforms one-shot generation.

✓ **Takeaway.** Node-by-node formalization of external knowledge into executable SCMs outperforms the one-shot method.

4.3 Causal Query Generation and Answer Verification

Query Generation. After generating the SCMs with our framework, we sample causal queries and execute the SCMs to obtain ground-truth answers. Because our goal is to measure and improve causal reasoning itself, we use discrete $\mathbf{V}$ and $\mathbf{U}$ to avoid conflating causal reasoning with the additional mathematical skills required in continuous probabilistic inference. Instead of predicting only the queried target—which risks exploiting the spurious effect of correctly guessing a single target node—the training LLMs predict all endogenous variables for deduction, intervention, and counterfactual queries; all exogenous variables for abduction queries; and the unobserved subset of $\mathbf{U}$ for counterfactual queries.

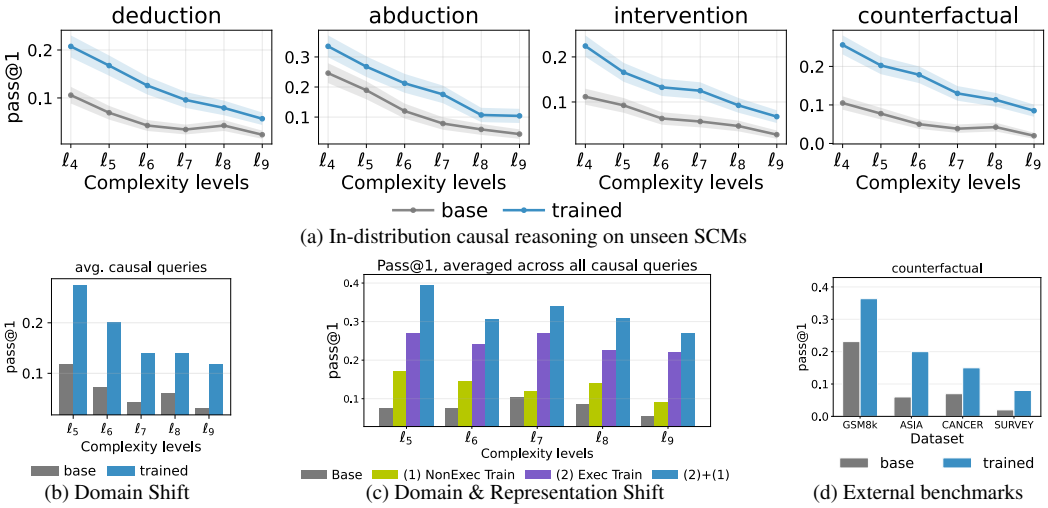
Answer Verification. Since structural mechanisms may not be invertible, the same observation may be compatible with multiple assignments of $\mathbf{U}$. We therefore enumerate candidate assignments up to a fixed budget and retain those that reproduce the observation. We apply the intervention and execute the modified SCM under each compatible assignment, producing the set of valid counterfactual outcomes. A prediction is considered correct if it matches any outcome in this set. Additional details and discussion of these design choices are provided in Appendix I.

5 Experiments

The main claim of our work is that verifiable supervision from scalable causal simulators can improve LLM causal reasoning over scales and representations. We evaluate this claim through five complementary studies:

Section	Goal
§5.1 Generalization	Transfer causal reasoning across unseen domains and representations
§5.2 Scaling	Leverage scalable simulators to study curriculum and data scaling
§5.3 Self-Improvement	Improve reasoning through self-generated causal simulators
§5.4 Data Augmentation	Formalize real-world causal knowledge for data augmentation
§5.5 Performance Anchoring	Contextualize accuracy against larger and frontier reference models

General Setup. Unless otherwise noted, the following setup is held constant across experiments. **Data.** We construct executable SCMs with four-letter variable names (NONSENSE) and medical concepts (MEDICAL) using `GPT-5`. We expand each SCM node-by-node up to 10 endogenous variables and retain every incremental version. These executable *code* SCMs are then converted



(a) In-distribution causal reasoning on unseen SCMs

(b) Domain Shift

(c) Domain & Representation Shift

(d) External benchmarks

Figure 4: **Improving Causal Reasoning Across Distributions**

into natural-language (*nl*) representations. Unless otherwise stated, training uses NONSENSE-*code*. MEDICAL-*code* and MEDICAL-*nl* are never trained on and solely used for evaluation.

Training. We fine-tune `Qwen2.5-3B-Instruct`, which we refer to as the BASE MODEL. We bootstrap causal reasoning using a STaR/RFT/RAFT-style rejection-sampling procedure [40, 41, 42]: for each causal query, the model generates a solution, which we verify against the ground-truth answer obtained by executing the SCM. Only correct solutions are retained for supervised fine-tuning. Training curricula use SCM sizes l_2, l_4, l_6, and l_8, where l_k denotes the incremental SCM version containing k endogenous variables. The curriculum and amount of training data vary by experiment which we detail below.

Evaluation. For our synthetic datasets, we use held-out NONSENSE-*code*, MEDICAL-*code*, and MEDICAL-*nl* SCMs. Each contains 6,000 intervention and counterfactual queries spanning 20 SCM IDs per topology, 10 incremental scales, and five queries per type. External benchmarks and the NIHSS augmentation study use separate evaluation sets, described in their respective sections. Examples and full dataset details are provided in Appendices A.7 and A.5. We use Pass@1 as the primary metric and report additional Pass@k and majority-vote results in Appendix F.3.

5.1 Improving Causal Reasoning Across Distributions

Motivation. We first ask whether training on executable causal simulators improves causal reasoning beyond the training distribution. We evaluate progressively larger shifts in semantics and representation, followed by evaluation on independently constructed external benchmarks.

Method. Unless otherwise stated, we evaluate the best-performing NONSENSE-*code* model from our scaling experiments (Sec. 5.2), trained with the BLOCK curriculum and 135 queries per type. We compare it against the BASE MODEL across four evaluation settings: ▶ **In-Distribution:** held-out NONSENSE-*code* SCMs; ▶ **Domain Shift:** MEDICAL-*code* SCMs, which change the semantic content while preserving the code representation; ▶ **Domain & Representation Shift:** MEDICAL-*nl* SCMs, which change both semantics and representation; and ▶ **External Benchmarks:** independently constructed causal problems derived from GSM8K [24] and the ASIA, CANCER, and SURVEY Bayesian networks from `bnlearn` [39].

For the domain and representation shift experiment, we test how the training representation affects transfer. We compare three models: (i) one trained on NONSENSE-*code*, (ii) one trained directly on NONSENSE-*nl*, and (iii) the NONSENSE-*code* model further fine-tuned on NONSENSE-*nl*. All three are evaluated on the same held-out MEDICAL-*nl* set. To further evaluate generalization, we evaluate the NONSENSE-*code* model on partially specified, verbalized GSM8K SCMs in Appendix F.2.

Results. ▶ **In-Distribution:** The fine-tuned model consistently outperforms the BASE MODEL on held-out NONSENSE-*code* SCMs (Fig. 4a). ▶ **Domain Shift:** These gains persist on MEDICAL-*code* SCMs (Fig. 4b). ▶ **Domain & Representation Shift:** Improvements also transfer to MEDICAL-*nl* SCMs (Fig. 4c). The model trained on NONSENSE-*code* outperforms the model trained directly on NONSENSE-*nl*, and further fine-tuning the code-trained model on NONSENSE-*nl* yields the strongest

performance. This suggests that training on executable representations provides a foundation that transfers across representations, while representation-specific training provides additional gains. ► **External Benchmarks:** The NONSENSE-*code* model also improves performance on the independently constructed external benchmarks (Fig. 4d). We observe a similar improvement when evaluating on partially specified verbalized GSM8K SCMs (Table 33).

✓ **Takeaway.** Training on executable causal queries improves LLM causal reasoning on unseen SCMs and generalizes across semantic and representation shifts.

5.2 Scaling Causal Reasoning via Complexity and Dataset Size

Motivation. A benefit of our framework is that we can scale both the complexity of the SCMs and the amount of data they generate. We use the this to study two questions: if progressively increasing SCM complexity improves learning, and if causal reasoning improves with more training data.

Method. ► **Curriculum:** In BLOCK, training proceeds iteratively: at each SCM size l_k, the current model generates solutions to causal queries, is fine-tuned on the correct solutions, and the resulting model is then used to generate solutions at the next level. In contrast, AT-ONCE uses the initial model to generate solutions for all complexity levels upfront, after which fine-tuning proceeds on the correct solutions through the levels in increasing order. EASYONLY trains only on l_2, and HARDONLY trains only on l_8. To isolate the effect of curriculum, all four settings use the same total budget of 64,800 intervention and counterfactual training queries. ►**Dataset Size.** We then fix the BLOCK curriculum and vary the amount of generated supervision, training separate models with 5, 15, 45, and 135 queries per type at each SCM version. All models are evaluated on the same held-out NONSENSE-*code* set described in the General Setup.

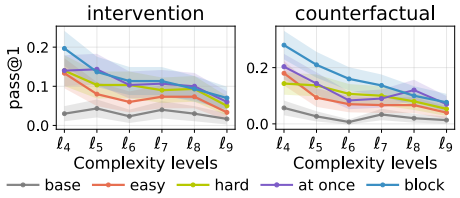


Figure 5: **Curriculum Matters.**

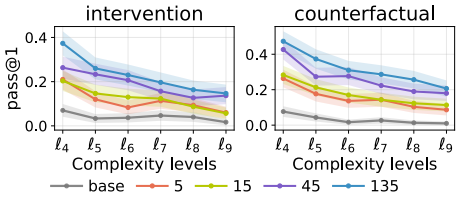


Figure 6: **More Data = Improved Performance**

Results. ► **Curriculum Matters.** (Fig. 5) The BLOCK curriculum outperforms all other schemes. This confirms that progressively increasing SCM size improves learning. ► **More Data = Improved Performance.** (Fig. 6) Scaling up the number of causal queries during training consistently yields higher accuracy across query types. See App R.5 & R.4 for additional results. We perform an ablation varying the number of distinct SCMs versus the number of queries, see App. R.6.

✓**Takeaway.** Causal reasoning improves with a progressive curriculum over increasingly complex SCMs and greater amounts of generated supervision.

5.3 Self-Improvement

Motivation. We next ask whether an LLM can bootstrap its causal reasoning ability using SCMs that it generates itself, and how the effectiveness of this self-generated data compares with data generated by stronger models.

Method. We construct three NONSENSE-*code* datasets using `GPT-5`, `GPT-5-mini`, and `Qwen2.5-7B-Instruct`. For each generator, we produce 180 SCM IDs and train `Qwen2.5-7B-Instruct` using the BLOCK curriculum with 15 intervention and 15 counterfactual queries per level. In the final setting, the same `Qwen2.5-7B-Instruct` model therefore serves as both the SCM generator and the model being improved. All three models are evaluated on the common held-out NONSENSE-*code* test set described above.

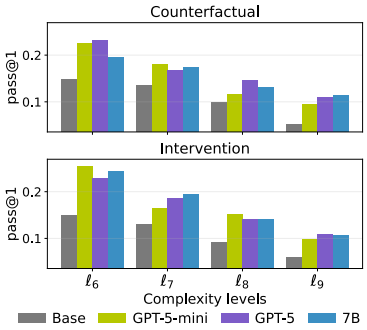


Figure 7: **Self-Improvement**

Results. All three generator sources yield clear improvements over the corresponding base 7B model(Fig. 7). Performance is also similar across SCMs generated by `GPT-5`, `GPT-5-mini`, and `Qwen2.5-7B-Instruct` itself. Thus, within the scale, representation, and domain studied here,

self-generated SCMs provide sufficient training signal for the model to improve its own causal reasoning. Additional results are provided in Appendix D.

✓ **Takeaway.** An LLM can improve its causal reasoning using SCM that it generates itself.

5.4 Case-Study: Augmenting Existing Data

Motivation. A benefit of our framework is that if there exits external knowledge thatcan be formalized as an executable SCM, the resulting simulator can generate additional interventional and counterfactual training data beyond what is originally available.

Method. We use the NIH Stroke Scale (NIHSS) as a case study and formalize its clinical scoring rules into an executable SCM using our procedure from Sec. 4.2. We then compare two training-data regimes over SCM sizes: a *Small* dataset with 50 intervention and 50 counterfactual queries per size (400 queries total), and an *Augmented* dataset with 1,000 queries per type per size (8,000 queries total). Both models use the same BLOCK training procedure and are evaluated on a separate held-out NIHSS set spanning SCM sizes l_1–l_{10} with 400 queries per type per size.

Results. The model trained on the augmented dataset consistently outperforms the small-data model across SCM sizes (Fig. 8), with the improvement becoming larger as SCM complexity increases.

✓ **Takeaway.** Formalizing non-executable causal knowledge into an executable SCM enables data augmentation that improves causal reasoning in the original domain.

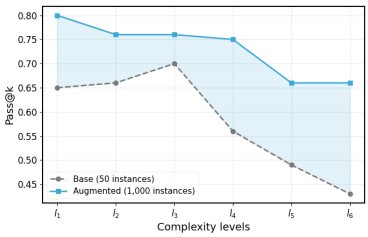


Figure 8: **Data Augmentation**

5.5 Anchoring Absolute Performance

Motivation. A natural question is how does this performance compare to larger and frontier models.

Method. We compare the trained 3B model with its untrained checkpoint, larger/newer Qwen models, and GPT-5.5 on the same held-out code-domain sets (Table 3). Appendix B.1 gives query-type results and the reported protocol.

Table 3: Pass@1 (%). N: NONSENSE-*code*; M: MEDICAL-*code*.

Model	N	M
Qwen2.5-3B, base	14.73	18.26
Qwen2.5-3B, trained	**39.18**	**31.07**
Qwen2.5-7B, base	22.48	28.90
Qwen3-4B, base	25.83	39.59
GPT-5.5	44.73	60.88

Results. The trained model exceeds the base 7B model in both domains and reaches 39.18% versus GPT-5.5's 44.73% on NONSENSE-*code*. The larger medical-domain gap (31.07% versus 60.88%) highlights remaining transfer limitations. These are performance anchors, not compute-matched comparisons or evidence that causal reasoning is solved. We verify that the training procedure transfers across model families, obtaining substantial improvements by training Llama 3.2 3B with the same framework (Appendix C).

6 Discussions

We introduce **CauSim**, a novel framework that builds *increasingly complex causal simulators*, unlocking executable supervision across scales and representations. Our key innovation is the *incremental* growth of these simulators, enabling globally complex causal reasoning environments that would be difficult to construct otherwise. **CauSim** operates across representations, *informalizing* executable representations in non-executable ones, which we instantiate as verbalized graphs, and *formalizing* non-executable representations into executable code. Our study of **CauSim** reveals its utility for improving causal reasoning: generalization to unseen domains and representations, consistent performance gains from curriculum and data scaling, LLM self-improvement through self-generated simulators, and data augmentation.

► **Limitations.** While **CauSim** demonstrates the promise of increasingly complex causal simulators as verifiable training environments, two limitations should be noted. First, although we demonstrate generalization across several settings, this does not guarantee models acquire fully invariant causal mechanisms (see App. K). Second, the limits of scaling simulator-generated supervision remain empirical. Within the range studied, we observe no plateau as additional training data is introduced, and our generator ablation suggests that generator quality is not yet the dominant bottleneck. However, further gains will ultimately depend on factors such as model capacity, the diversity and quality of

generated SCMs, and alignment between the training and evaluation distributions. We discuss these limits of recursive self-improvement and generator support further in Appendix L.

► **Future Work.** We see three promising directions. First, our current informalization instantiates non-executable representations as natural language, but the broader vision extends to richer causal systems: for instance, 2D/3D environments grounded by SCMs could enable spatial causal reasoning grounded with SCM supervison (see Table 36). Second, we assume an explicit SCM as input; future work could draw on LLM parametric knowledge or elicit missing causal specifications through active task disambiguation [43] or amortized sequential experimental design [44]. Third, we focus primarily on deterministic settings. Future work should extend to stochastic causal reasoning, better capturing human-like reasoning under uncertainty where counterfactuals are ambiguous.

ACKNOWLEDGMENTS

We thank the anonymous NeurIPS reviewers and members of the van der Schaar lab for their insightful comments and suggestions. NA thanks W.D. Armstrong Trust for their support. NA's PhD studentship is funded by the Cystic Fibrosis Trust. AK's PhD studentship is funded by Sanofi. This work was supported by Azure sponsorship credits granted by Microsoft's AI for Good Research Lab.

References

[1] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651:607–613, 2026. doi: 10.1038/s41586-025-09833-y. URL https://doi.org/10.1038/s41586-025-09833-y. Published online 12 November 2025.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[3] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

[4] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728963. URL https://doi.org/10.1145/3728963.

[5] Xiaoliang Luo, Akilles Rechardt, Guangzhi Sun, Kevin K. Nejad, Felipe Yáñez, Bati Yilmaz, Kangjoo Lee, Alexandra O. Cohen, Valentina Borghesani, Anton Pashkov, Daniele Marinazzo, Jonathan Nicholas, Alessandro Salatiello, Ilia Sucholutsky, Pasquale Minervini, Sepehr Razavi, Roberta Rocca, Elkhan Yusifov, Tereza Okalova, Nianlong Gu, Martin Ferianc, Mikail Khona, Kaustubh R. Patil, Pui-Shee Lee, Rui Mata, Nicholas E. Myers, Jennifer K. Bizley, Sebastian Musslick, Isil Poyraz Bilgin, Guiomar Niso, Justin M. Ales, Michael Gaebler, N. Apurva Ratan Murty, Leyla Loued-Khenissi, Anna Behler, Chloe M. Hall, Jessica Dafflon, Sherry Dongqi Bao, and Bradley C. Love. Large language models surpass human experts in predicting neuroscience results. *Nature Human Behaviour*, 9(2):305–315, 2025. doi: 10.1038/s41562-024-02046-9. URL https://doi.org/10.1038/s41562-024-02046-9. Published online 27 November 2024.

[6] Zhijing Jin, Yuen Chen, Felix Leeb, Luigi Gresele, Ojasv Kamal, Zhiheng Lyu, Kevin Blin, Fernando Gonzalez Adauto, Max Kleiman-Weiner, Mrinmaya Sachan, et al. Cladder: Assessing

causal reasoning in language models. *Advances in Neural Information Processing Systems*, 36: 31038–31065, 2023.

[7] Swagata Ashwani, Kshiteesh Hegde, Nishith Reddy Mannuru, Dushyant Singh Sengar, Mayank Jindal, Krishna Chaitanya Rao Kathala, Dishant Banga, Vinija Jain, and Aman Chadha. Cause and effect: can large language models truly understand causality? In *Proceedings of the AAAI Symposium Series*, volume 4, pages 2–9, 2024.

[8] Haoang Chi, He Li, Wenjing Yang, Feng Liu, Long Lan, Xiaoguang Ren, Tongliang Liu, and Bo Han. Unveiling causal reasoning in large language models: Reality or mirage? In *Advances in Neural Information Processing Systems*, volume 37, pages 96640–96670, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/af2bb2b2280d36f8842e440b4e275152-Abstract-Conference.html.

[9] Matej Zečević, Moritz Willig, Devendra Singh Dhami, and Kristian Kersting. Causal parrots: Large language models may talk causality but are not causal. *arXiv preprint arXiv:2308.13067*, 2023.

[10] Paul W. Holland. Statistics and causal inference. *Journal of the American Statistical Association*, 81(396):945–960, 1986. doi: 10.1080/01621459.1986.10478354.

[11] Miguel A. Hernán and James M. Robins. *Causal Inference: What If*. Chapman & Hall/CRC, Boca Raton, 2020.

[12] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, UK, 2 edition, 2009. doi: 10.1017/CBO9780511803161.

[13] Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *arXiv preprint arXiv:2505.03335*, 2025. URL https://arxiv.org/abs/2505.03335.

[14] Elias Bareinboim and Judea Pearl. Causal inference and the data-fusion problem. *Proceedings of the National Academy of Sciences*, 113(27):7345–7352, 2016. doi: 10.1073/pnas.1510507113. URL https://www.pnas.org/doi/abs/10.1073/pnas.1510507113.

[15] Judea Pearl. Causal diagrams for empirical research. *Biometrika*, 82(4):669–688, 1995. doi: 10.1093/biomet/82.4.669.

[16] Zhijing Jin, Jiarui Liu, Zhiheng Lyu, Spencer Poff, Mrinmaya Sachan, Rada Mihalcea, Mona Diab, and Bernhard Schölkopf. Can large language models infer causation from correlation? *arXiv preprint arXiv:2306.05836*, 2023.

[17] Zeyu Wang. CausalBench: A comprehensive benchmark for evaluating causal reasoning capabilities of large language models. In *Proceedings of the 10th SIGHAN Workshop on Chinese Language Processing (SIGHAN-10)*, pages 143–151, Bangkok, Thailand, August 2024. Association for Computational Linguistics. URL https://aclanthology.org/2024.sighan-1.17/.

[18] Yuefei Chen, Vivek K. Singh, Jing Ma, and Ruixiang Tang. Counterbench: Evaluating and improving counterfactual reasoning in large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 30350–30358, 2026. doi: 10.1609/aaai.v40i36.40287. URL https://ojs.aaai.org/index.php/AAAI/article/view/40287.

[19] Jacqueline Maasch, John Kalantari, and Kia Khezeli. Causalarc: Abstract reasoning with causal world models. *arXiv preprint arXiv:2509.03636*, 2025.

[20] Jacqueline RMA Maasch, Alihan Hüyük, Xinnuo Xu, Aditya V Nori, and Javier Gonzalez. Compositional causal reasoning evaluation in language models. *arXiv preprint arXiv:2503.04556*, 2025.

[21] Javier González and Aditya Nori. Does reasoning emerge? examining the probabilities of causation in large language models. *Advances in Neural Information Processing Systems*, 37: 117737–117761, 2024.

[22] Jiaxuan Li, Lang Yu, and Allyson Ettinger. Counterfactual reasoning: Testing language models' understanding of hypothetical scenarios. *arXiv preprint arXiv:2305.16572*, 2023.

[23] Yian Li, Wentao Tian, Yang Jiao, Jingjing Chen, Tianwen Qian, Bin Zhu, Na Zhao, and Yu-Gang Jiang. Look before you decide: Prompting active deduction of mllms for assumptive reasoning, 2024. URL https://arxiv.org/abs/2404.12966. Version 5, last revised 17 Apr 2025.

[24] Xinnuo Xu, Rachel Lawrence, Kshitij Dubey, Atharva Pandey, Risa Ueno, Fabian Falck, Aditya V Nori, Rahul Sharma, Amit Sharma, and Javier Gonzalez. Re-imagine: Symbolic benchmark synthesis for reasoning evaluation. *arXiv preprint arXiv:2506.15455*, 2025.

[25] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. doi: 10.48550/arXiv.2110.14168.

[26] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 2015. PMLR. URL https://proceedings.mlr.press/v37/schulman15.html.

[27] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[28] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[29] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[30] Xumeng Wen, Zihan Liu, Shun Zheng, Shengyu Ye, Zhirong Wu, Yang Wang, Zhijian Xu, Xiao Liang, Junjie Li, Ziming Miao, Jiang Bian, and Mao Yang. Reinforcement learning with verifiable rewards implicitly incentivizes correct reasoning in base llms. *arXiv preprint arXiv:2506.14245*, 2025.

[31] Xiaoyin Chen, Jiarui Lu, Minsu Kim, Dinghuai Zhang, Jian Tang, Alexandre Piché, Nicolas Gontier, Yoshua Bengio, and Ehsan Kamalloo. Self-evolving curriculum for llm reasoning. *arXiv preprint arXiv:2505.14970*, 2025.

[32] Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Step-by-step optimization-like reasoning in LLMs over expanding search spaces, 2026. URL https://arxiv.org/abs/2606.05464. arXiv preprint arXiv:2606.05464.

[33] Xiao Liu, Da Yin, Chen Zhang, Dongyan Zhao, and Yansong Feng. Eliciting and improving the causal reasoning abilities of large language models with conditional statements. *Computational Linguistics*, 51:467–504, June 2025. doi: 10.1162/coli_a_00548. URL https://aclanthology.org/2025.cl-2.3/.

[34] Alihan Hüyük, Xinnuo Xu, Jacqueline Maasch, Aditya V Nori, and Javier González. Reasoning elicitation in language models via counterfactual feedback. *arXiv preprint arXiv:2410.03767*, 2024.

[35] Aniket Vashishtha, Qirun Dai, Hongyuan Mei, Amit Sharma, Chenhao Tan, and Hao Peng. Executable counterfactuals: Improving llms' causal reasoning through code. *arXiv preprint arXiv:2510.01539*, 2025.

[36] Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason E. Weston. Self-rewarding language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 57905–57923. PMLR, 2024. URL https://proceedings.mlr.press/v235/yuan24d.html.

[37] Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search, 2024. URL https://arxiv.org/abs/2408.08152.

[38] Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela Van Der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 1864–1886. PMLR, 2025. URL https://proceedings.mlr.press/v267/astorga25a.html.

[39] Marco Scutari. Learning bayesian networks with the bnlearn r package. *Journal of Statistical Software*, 35(3):1–22, 2010. doi: 10.18637/jss.v035.i03. URL https://www.jstatsoft.org/article/view/v035i03.

[40] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, 2022.

[41] Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*, 2023. doi: 10.48550/arXiv.2308.01825. URL https://arxiv.org/abs/2308.01825.

[42] Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. RAFT: Reward ranked finetuning for generative foundation model alignment. *Transactions on Machine Learning Research*, 2023. doi: 10.48550/arXiv.2304.06767. URL https://arxiv.org/abs/2304.06767.

[43] Katarzyna Kobalczyk, Nicolás Astorga, Tennison Liu, and Mihaela van der Schaar. Active task disambiguation with LLMs. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=JAMxRSXLFz.

[44] Anita Kriz, Antonin Berthon, Nicolás Astorga, and Mihaela van der Schaar. Amortizing sequential experimental design in reasoning LLMs, 2026. URL https://arxiv.org/abs/2610.20832. arXiv preprint arXiv:2610.20832.

[45] David Galles and Judea Pearl. An axiomatic characterization of causal counterfactuals. *Foundations of Science*, 3(1):151–182, 1998. doi: 10.1023/A:1009602825894.

[46] Ilya Shpitser and Judea Pearl. Complete identification methods for the causal hierarchy. *Journal of Machine Learning Research*, 9:1941–1979, 2008.

[47] Jonas Peters, Dominik Janzing, and Bernhard Schölkopf. *Elements of Causal Inference: Foundations and Learning Algorithms*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, MA, USA, 2017.

[48] Bernhard Schölkopf, Dominik Janzing, Jonas Peters, Eleni Sgouritsa, Kun Zhang, and Joris M. Mooij. On causal and anticausal learning. In *Proceedings of the 29th International Conference on Machine Learning*, pages 1255–1262. Omnipress, 2012.

[49] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A. Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2(11):665–673, 2020. doi: 10.1038/s42256-020-00257-z.

[50] Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021. doi: 10.1109/JPROC.2021.3058954.

[51] Jonas Peters, Peter Bühlmann, and Nicolai Meinshausen. Causal inference by using invariant prediction: Identification and confidence intervals. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 78(5):947–1012, 2016. doi: 10.1111/rssb.12167.

[52] Mateo Rojas-Carulla, Bernhard Schölkopf, Richard E. Turner, and Jonas Peters. Invariant models for causal transfer learning. *Journal of Machine Learning Research*, 19(36):1–34, 2018.

[53] Martin Arjovsky, Léon Bottou, Ishaan Gulrajani, and David Lopez-Paz. Invariant risk minimization. *arXiv preprint arXiv:1907.02893*, 2019.

[54] Francesco Locatello, Stefan Bauer, Mario Lucic, Gunnar Rätsch, Sylvain Gelly, Bernhard Schölkopf, and Olivier Bachem. Challenging common assumptions in the unsupervised learning of disentangled representations. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4114–4124. PMLR, 2019.

[55] Jesse Vig, Sebastian Gehrmann, Yonatan Belinkov, Sharon Qian, Daniel Nevo, Yaron Singer, and Stuart Shieber. Investigating gender bias in language models using causal mediation analysis. In *Advances in Neural Information Processing Systems*, volume 33, pages 12388–12401, 2020.

[56] Atticus Geiger, Hanson Lu, Thomas Icard, and Christopher Potts. Causal abstractions of neural networks. In *Advances in Neural Information Processing Systems*, volume 34, pages 9574–9586, 2021.

[57] Qizhe Xie, Minh-Thang Luong, Eduard Hovy, and Quoc V. Le. Self-training with noisy student improves imagenet classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10687–10698, 2020.

[58] Jiaxin Huang, Shixiang Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1051–1068, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.67. URL `https://aclanthology.org/2023.emnlp-main.67/`.

[59] Eric Arazo, Diego Ortego, Paul Albert, Noel E. O'Connor, and Kevin McGuinness. Pseudo-labeling and confirmation bias in deep semi-supervised learning. In *2020 International Joint Conference on Neural Networks*, pages 1–8. IEEE, 2020. doi: 10.1109/IJCNN48605.2020.9207304.

[60] Baixu Chen, Junguang Jiang, Ximei Wang, Pengfei Wan, Jianmin Wang, and Mingsheng Long. Debiased self-training for semi-supervised learning. In *Advances in Neural Information Processing Systems*, 2022.

[61] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. AI models collapse when trained on recursively generated data. *Nature*, 631:755–759, 2024. doi: 10.1038/s41586-024-07566-y.

[62] Sina Alemohammad, Josue Casco-Rodriguez, Lorenzo Luzi, Ahmed Imtiaz Humayun, Hossein Babaei, Daniel LeJeune, Ali Siahkoohi, and Richard G. Baraniuk. Self-consuming generative models go MAD. In *International Conference on Learning Representations*, 2024.

[63] Matthias Gerstgrasser, Rylan Schaeffer, Apratim Dey, Rafael Rafailov, Henry Sleight, John Hughes, Tomasz Korbak, Rajashree Agrawal, Dhruv Pai, Andrey Gromov, Daniel A. Roberts, Diyi Yang, David L. Donoho, and Sanmi Koyejo. Is model collapse inevitable? breaking the curse of recursion by accumulating real and synthetic data. In *Conference on Language Modeling*, 2024.

[64] Paulius Rauba and Mihaela van der Schaar. Deep hierarchical learning with nested subspace networks for large language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL `https://openreview.net/forum?id=ymUOPsbxLi`.

Appendix Index

Link	What to look for	Page
Appendix A	**Experimental details, prompts, and examples.** Construction, verification, translation, dataset configurations, and representative SCMs.	17
Appendix B	**Absolute-performance anchoring.** Comparisons against the base, larger/newer Qwen models, and GPT-5.5.	28
Appendix B.1	Reported evaluation protocol and query-type results.	28
Appendix C	**Generalization across model families.** Llama 3.2 3B results across domains and representations.	29
Appendix D	**7B generator comparisons.** Self-generated, GPT-5-mini, and GPT-5 training data, including delta-highlighted results.	29
Appendix E	**Informal representation experiments.** Medical and nonsensical natural-language SCMs, with topology-specific results.	32
Appendix F	**Additional benchmarks and diagnostics.** Formalized GSM8K, bnlearn networks, and one-shot generation failures.	36
Appendix F.2	Partially specified, verbalized GSM8K evaluation.	36
Appendix F.3	External-benchmark Pass@k and majority-vote results.	37
Appendix G	**Why causal simulators?** Domain examples and meaningful causal queries.	38
Appendix H	**LLM-based SCM generation.** Design rationale and procedural-generator baselines.	39
Appendix I	**Discrete SCMs.** Finite-support verification and the distinction from continuous inference.	41
Appendix J	**Related-work comparison.** Capabilities of prior SCM-based approaches.	41
Appendix K	**Pattern recognition versus mechanism learning.** Scope of causal-query accuracy and tests of mechanistic reasoning.	42
Appendix L	**Limits of self-improvement.** One-step bootstrapping, generator support, and recursive-training limitations.	43
Appendix M	**Incremental SCM construction.** Full procedure and pseudocode.	44
Appendix N	**Generation prompts.** Instantiation, topology, planning, and localized execution templates.	46
Appendix O	**Evaluation prompts.** Counterfactual and intervention examples.	51
Appendix P	**Training details.** Pipeline, SFT settings, hyperparameter selection, and LoRA configurations.	57
Appendix Q	**Data generation.** Support-valued semantics, trace sampling, query construction, and prompt assembly.	59
Appendix R	**Additional results.** Query types, NIHSS augmentation, curriculum, data volume, and SCM/query allocation.	63

A Experimental Details, Prompts, and Examples

This section provides additional details for SCM construction and translation, the programmatic verification used to retain generated SCMs, the experimental configurations used in Sec. 5, and representative examples of the datasets used for training and evaluation.

A.1 Incremental SCM Construction

CauSim constructs SCMs incrementally using four stages: specification, planning, execution, and verification. We first illustrate the procedure with a running example and then provide the corresponding prompt templates.

A.1.1 Running Example

Some inputs and outputs below are abbreviated for readability.

Step 1: Specification. The specification stage creates an initial executable SCM. For example, given

```
Nodes: 1
Domain: math
```

the model may produce

```
def U_A():
    return random()

def f_A(u_a):
    return 2.0 * u_a

def run_once(seed):
    u_a = U_A()
    A = f_A(u_a)
    return {"A": A}
```

Step 2: Planning. To expand the SCM, the model receives a semantic view of the current SCM rather than the complete program. For example,

```
{
  "summary":
    "Exogenous: U_A | Structural: f_A |
     Output: A | Edge: U_A -> f_A",

  "U_A": {
    "description": "Samples the exogenous input for A.",
    "children": ["f_A"]
  },

  "f_A": {
    "description": "Computes A from its exogenous input.",
    "parents": ["U_A"]
  }
}
```

The planner proposes one local graph update:

```
{
  "new_function": "f_B",
  "new_exo_noise": "U_B",
  "direct_parents": ["f_A"],
  "direct_children": []
}
```

Step 3: Execution. The execution stage receives the plan together with only the relevant local SCM context and implements the update:

```
def U_B():
    return random()

def f_B(A, u_b):
    return A + u_b

def run_once(seed):
    u_a = U_A()
    A = f_A(u_a)
    u_b = U_B()
    B = f_B(A, u_b)
    return {"A": A, "B": B}
```

Step 4: Verification. The resulting program is subjected to deterministic structural and execution checks described in Sec. A.2. Invalid candidates are rejected and regenerated up to the specified retry budget. Steps 2–4 repeat until the requested SCM size is reached.

A.1.2 Specification Prompt

For the NONSENSE domain, the initial SCM is generated using:

```
You are generating an EXECUTABLE, STOCHASTIC Structural
Causal Model (SCM) in Python.

HARD REQUIREMENTS:
- All variables used in the SCM must have no semantic
  meaning. Variable names should be non-existent words
  that are short, i.e., within 5 characters.
- Functions may use if/else, for-loops, boolean logic,
  and string comparisons. DO NOT make the SCM mathematical.
- Allowed imports: random, typing. No others.
- Use docstrings ONLY (no # comments).
- Define exogenous samplers named U_* (pure, no I/O).
  - Each U_* must be independent of all other
    exogenous variables.
  - There must be no exogenous confounding.
- Define structural assignments named f_* (pure, no I/O).
  - Each f_* must have ONE exogenous parent U_*.
  - Each f_* must take all parent variables explicitly
    as arguments.
  - Structural functions are deterministic given inputs.
  - The SCM MUST be acyclic.
  - The SCM MUST be connected.
- Include a driver:
    def run_once(seed: int | None = None)
  - If seed is not None, call random.seed(seed).
  - Assign each function call on a separate line.
  - Return ONLY outputs of structural assignments f_*.

OUTPUT FORMAT:
1) All U_* sampler definitions
2) All f_* structural function definitions
3) The run_once(seed) driver

Return ONLY a single Python code block. No prose.

Task:
- Create an SCM with [NUM_NODES] exogenous samplers and
  [NUM_NODES] structural functions.
- The internal nodes must follow a [TOPOLOGY] topology.

[TOPOLOGY DEFINITION]
```

For MEDICAL SCMs, we use the corresponding logic-based variant:

```
You are generating a SMALL, EXECUTABLE, STOCHASTIC
Structural Causal Model (SCM) in Python.

This SCM must be logic-based, symbolic, or mechanistic,
based on a clinical, biological, or medical domain.
DO NOT make the SCM mathematical.

Functions may return booleans, categorical strings,
or symbolic states.

HARD REQUIREMENTS:
- Allowed imports: random, typing. No others.
- Use docstrings ONLY.
```

```
- Define independent exogenous samplers named U_*.
- Define structural assignments named f_*.
- Each f_* must take all parent variables explicitly.
- Each f_* must have ONE exogenous parent.
- The SCM must be acyclic.
- Structural functions are deterministic given inputs.
- Include a run_once(seed) driver that returns outputs
  from f_* only.

OUTPUT FORMAT:
1) All U_* exogenous sampler definitions
2) All f_* structural mechanism definitions
3) The run_once(seed) driver

Return ONLY a single Python code block. No prose.

Task:
- Create an SCM with [NUM_NODES] exogenous samplers and
  [NUM_NODES] structural functions.
- The internal nodes must follow a [TOPOLOGY] topology.

[TOPOLOGY DEFINITION]
```

A.1.3 Topology Constraints

Our main experiments use chain, inverted-star, and layered endogenous graph structures.

```
A chain topology is a single linear sequence of
internal nodes.

Rules:
- Exactly one parent per node except the first.
- Exactly one child per node except the last.
- Shape: A -> B -> C -> ...
- If inserting a new node between X and Y:
  - Y removes X as a parent.
  - Y adds the new node as its only parent.
  - The new node takes X as its parent.
```

```
An inverted star topology is the reverse of a star:
many parents converge into a single child.

Rules:
- A single sink node has multiple parents.
- All parents have exactly one child: the sink.
- The sink has no children.
- Parents do not depend on each other.
- Shape: {P1, P2, P3, ...} -> Child.
- New nodes can only be inserted as direct parents
  of the sink.
```

```
Rules:
- Every node belongs to exactly one layer.
- Multiple nodes may occupy one layer, but there
  are no edges within a layer.
- Edges are permitted only from layer k to layer k+1.
- Edges may not skip layers or point backwards.
```

A.1.4 Planning Prompt

At each incremental step, the planner receives a semantic view of the existing SCM:

```
You are planning the NEXT INCREMENTAL GROWTH of an existing
EXECUTABLE, STOCHASTIC Structural Causal Model (SCM)
in Python.

You are provided a SEMANTIC VIEW of the existing SCM:
- Exogenous samplers: U_* (external nodes)
- Structural assignments: f_* (internal nodes)

TASK
- Propose ONE new structural assignment and its
  corresponding exogenous sampler.
- Define the direct parents and direct children
  of this new structural assignment.
```

```
HARD REQUIREMENTS:
- The corresponding exogenous sampler is REQUIRED.
- direct_parents and direct_children may contain ONLY
  pre-existing structural assignments f_*.
- At least one of direct_parents or direct_children
  must be non-empty.
- The update must not introduce cycles.
- New function names must be unique.

OUTPUT FORMAT
Return STRICT JSON:
{
  "rationale": "one-sentence design intent",
  "new_function": "name of the new f_* function",
  "new_exo_noise": "name of the new U_* function",
  "direct_parents": ["f_*", ...],
  "direct_children": ["f_*", ...]
}

Additional Topology HARD Requirement:
- The updated SCM must obey the [TOPOLOGY] topology.

[TOPOLOGY DEFINITION]

SEMANTIC VIEW:
[CURRENT SCM SEMANTIC VIEW]
```

A.1.5 Execution Prompt

The execution stage receives the decision, local code context, and current driver:

```
You will IMPLEMENT a small update to an existing
EXECUTABLE, STOCHASTIC structural causal model (SCM)
in Python.

You are given:

1. DECISION JSON:
   - rationale
   - new_function
   - new_exo_noise
   - direct_parents
   - direct_children

2. CONTEXT CODE:
   - For each direct child: full function
   - For each direct parent: semantic information

3. DRIVER FUNCTION:
   def run_once(seed: int | None)

TASK:
- Define the new exogenous noise sampler.
- Define the new structural function.
- Its arguments must contain its corresponding
  exogenous input and all direct structural parents.
- Structural logic must be deterministic given inputs.
- If direct children are specified, modify their
  signatures and mechanisms.
- Update the driver in topological order.

HARD REQUIREMENTS:
- Preserve purity.
- Do not emit unrelated functions.
- Do not emit direct-parent implementations.

DECISION JSON:
[DECISION JSON]

CONTEXT CODE:
[LOCAL CODE CONTEXT]

DRIVER FUNCTION:
[CURRENT DRIVER]
```

A.2 Programmatic SCM Verification

Verification is deterministic and does not use an LLM. Generated or updated SCMs are retained only after passing the required structural and execution checks. Failed candidates are rejected and regenerated up to the retry budget.

Program restrictions. We check that imports belong to an allowed set and reject unsafe or impure operations such as file access, external requests, subprocess calls, or other prohibited side effects.

Acyclicity and execution order. We extract the generated causal graph and perform a topological sort. Candidates for which a complete topological ordering cannot be obtained are rejected as cyclic.

Exogenous structure. The validator checks that an exogenous node points only to endogenous structural nodes, that each exogenous variable affects at most one structural node, and that an endogenous structural node has at most one exogenous parent. We additionally require that exogenous randomness reaches at least one structural variable used by the driver.

Topology. For experiments with a specified topology, the endogenous subgraph is checked against the requested structural family. In our main experiments, this includes chain, inverted-star, and layered structures.

Node count. When a target SCM size is specified, the number of endogenous and exogenous nodes is checked against the requested value.

Executability. Generated programs are dynamically loaded and required to define the driver function. We execute the resulting SCM under controlled exogenous values and reject programs that raise an exception.

Reference-graph agreement. For formalization settings in which an explicit source graph is available, we additionally compare the endogenous nodes and directed edges of the generated SCM against the source graph. Missing or additional nodes and missing or spurious edges cause validation to fail.

Together, these checks ensure that retained SCMs are executable, acyclic, structurally consistent with the requested graph constraints, and suitable for mechanically computing causal-query answers.

A.3 SCM Translation and Formalization

CauSim translates between executable and non-executable causal representations. We distinguish *informalization*, which converts an executable SCM into natural language, from *formalization*, which converts external causal knowledge into an executable SCM.

A.3.1 Informalization: Executable SCM to Natural Language

We translate a structured representation of an executable SCM into semantically equivalent natural language:

```
You will be given a structured dictionary representation
of a Structural Causal Model (SCM).

YOUR TASK:
Translate the SCM into unstructured natural language
that is SEMANTICALLY EQUIVALENT and LOSSLESS.

NAME PRESERVATION:
All names MUST be preserved EXACTLY.

WHAT TO DESCRIBE:

1. Exogenous variables:
- State the variable's instantiation in the driver.
- Describe exactly how its value is generated.

2. Structural variables:
- State the variable's instantiation in the driver.
- Explicitly state all inputs in their original order.
- Describe exactly how its value is generated.

3. Driver output:
- Describe the return value of the driver.
- State every output name and the variable to which
  it corresponds.
```

```
OUTPUT RULES:
- One paragraph per variable in topological order.
- Output only unstructured natural language.
- Do not output code, equations, bullet points, or lists.
- Do not introduce assumptions or mechanisms not
  present in the SCM.

Dictionary:
[STRUCTURED SCM REPRESENTATION]
```

A.3.2 One-Shot Natural-Language Formalization

The one-shot baseline receives the full natural-language SCM in a single call:

```
You are given an unstructured natural language description
of a Structural Causal Model (SCM).

The description is guaranteed to be LOSSLESS and COMPLETE.

YOUR TASK:
Write the SCM as executable Python code that exactly
matches the described causal structure.

HARD REQUIREMENTS:
- Add imports: import random, import typing. No others.
- Use docstrings ONLY.
- Define one exogenous sampler per structural assignment.
- Each exogenous sampler must be independent.
- Define structural assignments named f_*.
- Each structural assignment must take all parent
  variables explicitly.
- Each structural assignment must have one
  exogenous parent.
- Structural assignments are deterministic given inputs.
- The SCM must be acyclic.
- Do not define additional functions.

OUTPUT FORMAT:
1) imports
2) all exogenous sampler definitions
3) all structural mechanism definitions
4) driver

Natural Language Description:
[NATURAL-LANGUAGE SCM]
```

A.3.3 Step-by-Step Formalization

Our step-by-step formalization uses a two-stage decision–action procedure.

```
You are given:

1. SEMANTIC INFORMATION FOR THE EXISTING SCM.
2. NEW ITEM that needs to be formalized.

YOUR TASK:
Insert the NEW ITEM into the existing SCM.

Specifically:
1. Choose the new structural function.
2. Choose its exogenous sampler.
3. Identify direct parents already in the SCM.
4. Identify direct children already in the SCM.

IMPORTANT:
If the NEW ITEM does not mention direct parents or
children in the current SCM, leave them empty.

Return Strict JSON:
{
  "new_function": "f_<item_name>",
  "new_exo_noise": "U_<item_name>",
  "direct_parents": ["f_<parent1>", ...],
  "direct_children": ["f_<child1>", ...]
}

SEMANTIC INFORMATION FOR THE EXISTING SCM:
[CURRENT SCM SEMANTIC INFORMATION]

NEW ITEM:
[NEW ITEM DESCRIPTION]
```

```
You will IMPLEMENT a small update to a structural causal
model in which ONE new structural function is added.

You are given:
1. A DECISION JSON.
2. The specification of the NEW FUNCTION.
3. CODE CONTEXT:
   - full code for each direct child
   - semantic information for each direct parent

TASK:
Implement exactly the update specified by the decision.

1. Define the new exogenous sampler.
2. Define the new structural function.
3. Modify each direct child if required.

OUTPUT FORMAT:
- Full definition of the new function.
- Full definition of the new exogenous sampler.
- Full definitions of modified direct children.
- Do not output prose.

DECISION JSON:
[DECISION JSON]

NEW FUNCTION DESCRIPTION:
[NEW ITEM]

CODE CONTEXT:
[LOCAL CODE CONTEXT]
```

A.3.4 NIH Stroke Scale Formalization

For NIHSS, the same decision–action procedure is augmented with the following domain constraints:

```
1. Treat each scored NIHSS item as a structural variable.
   Each score should be equally likely.

2. If an item explicitly depends on another item's value,
   that dependency MUST be represented in the function
   signatures.

3. Do not use "UN" unless specified in the item.

4. Do NOT rename, simplify, or reinterpret labels.

5. Update total_score to be the sum of ALL inputs.
   If an input is "UN", ignore it in the summation.
```

For example,

```
8. f_sensory
Test pinprick sensation.

Score:
0 = Normal; no sensory loss.
1 = Mild-to-moderate sensory loss.
2 = Severe or total sensory loss.

Patients with f_level_of_consciousness = 3
automatically score 2.
```

is formalized as

```
def f_sensory(f_level_of_consciousness, u_sensory):
    if f_level_of_consciousness == 3:
        return 2
    return u_sensory
```

A.3.5 Program-to-SCM Formalization for GSM8K

For GSM8K-derived tasks, the source representation is an executable reasoning program. We convert it into an SCM using:

```
You are given a Python program that solves a
reasoning problem.

Your task is to convert this program into a
Structural Causal Model written as executable Python code.

CORE PRINCIPLE:
Each variable is represented by an exogenous variable
and a deterministic structural function.

EXOGENOUS VARIABLES:
- Every directly assigned constant must have an
  exogenous variable.
- Its support must include the original value.

STRUCTURAL FUNCTIONS:
- Define one function per variable.
- Include all causal parents explicitly.
- Preserve the exact program logic.

DEPENDENCIES:
- The dependency structure must match the program exactly.

CONDITIONALS:
- Preserve all if/else logic.

LOOPS:
- Rewrite explicitly where possible or preserve them.

DRIVER:
- Execute variables in causal order.
- Return all endogenous variables.

INPUT PROGRAM:
[PROGRAM]
```

A.3.6 Bayesian-Network Formalization

For ASIA, CANCER, and SURVEY, the source representation is a fully specified Bayesian network with conditional probability distributions.

```
You are given a fully specified Bayesian Network in JSON.

The BN defines:
- variables
- conditional probability distributions (CPDs)

Your task is to convert the BN into an equivalent
executable Structural Causal Model.

- Use independent exogenous noise for stochastic variables.
- Structural functions are deterministic given their inputs.
- Select the correct CPD according to parent values.
- Implement stochastic CPDs using inverse-CDF sampling.
- Deterministic CPDs do not require exogenous noise.
- Execute variables in topological order.
- Return all endogenous variables.
- The SCM must reproduce the BN distribution.

Input BN:
[BAYESIAN NETWORK]
```

Step-by-Step BN Formalization. The decision stage is the same as above. The action stage is specialized to the supplied CPD:

```
You are given:

1. A DECISION JSON.
2. A NEW BN NODE SPECIFICATION containing the variable
   and its conditional probability table.
3. LOCAL CODE CONTEXT.

Implement exactly the specified graph update.

- Use independent exogenous noise for stochastic nodes.
- Use inverse-CDF sampling.
- Include every structural parent explicitly.
- Deterministic CPDs require no exogenous noise.
```

Dataset	Step-by-Step	One-Shot
NIHSS	**52/100**	20/100
ASIA	**84/100**	69/100
CANCER	**74/100**	51/100

Table 4: Step-by-step versus one-shot formalization accuracy (successful trials out of 100).

Experiment	Model	Training Data	Evaluation Data
In-distribution	Qwen2.5-3B-Instruct, BLOCK, 135 queries/type	GPT-5 NONSENSE-*code*	6,000 held-out NONSENSE-*code* queries
Domain shift	Same	Same	6,000 held-out MEDICAL-*code* queries
Domain + representation	Three Qwen2.5-3B-Instruct variants	NONSENSE-*code* and NONSENSE-*nl*	6,000 held-out MEDICAL-*nl* queries
External	Qwen2.5-3B-Instruct, trained	NONSENSE-*code*	GSM8K, ASIA, CANCER, and SURVEY
Partially specified	Qwen2.5-3B-Instruct, trained	NONSENSE-*code*	262 GSM8K SCMs, 500 total queries

Table 5: Experimental configurations for generalization.

```
- Preserve the supplied CPD exactly.

OUTPUT:
1. New exogenous function, if needed
2. New structural function
3. Modified direct-child functions, if needed

DECISION JSON:
[DECISION JSON]

NEW BN NODE SPECIFICATION:
[BN NODE AND CPD]

CODE CONTEXT:
[LOCAL CODE CONTEXT]
```

A.4 Formalization Results

Step-by-step formalization produces more valid executable SCMs than one-shot formalization for NIHSS and both Bayesian-network datasets:

A.5 Experimental Configurations

A.5.1 Generalization

For synthetic evaluation, each 6,000-query set contains three topologies, 20 SCM IDs per topology, 10 SCM sizes, five queries per type, and two query types. For the domain-and-representation experiment, we compare a NONSENSE-*code*-trained model, a directly NONSENSE-*nl*-trained model, and a code-trained model further fine-tuned on NONSENSE-*nl*. Formalized GSM8K contains 270 SCMs with 270 intervention and 264 counterfactual queries. ASIA, CANCER, and SURVEY each contain 100 intervention and 100 counterfactual queries.

A.5.2 Curriculum and Dataset Size

Training uses GPT-5-generated NONSENSE-*code* SCMs with chain, inverted-star, and layered topologies and 180 SCM IDs per topology. Training levels are l_2, l_4, l_6, and l_8. BLOCK and AT-ONCE use 15 queries per type at each level. EASYONLY uses 60 queries per type at l_2, while HARDONLY uses 60 queries per type at l_8.

A.5.3 Self-Improvement

Each generator produces 180 SCM IDs with incremental versions spanning 1–10 nodes. Training uses BLOCK over l_2, l_4, l_6, and l_8, with 15 intervention and 15 counterfactual queries per level.

A.5.4 NIHSS Data Augmentation

Both training conditions use BLOCK over l_2, l_4, l_6, and l_8. The evaluation set contains 400 queries per type at each size from l_1 through l_{10}.

Experiment	Model	Training Configuration	Evaluation
Curriculum	Qwen2.5-3B-Instruct	BLOCK, AT-ONCE, EASYONLY, HARDONLY; 64,800 queries each	Held-out NONSENSE-*code*
Dataset size	Qwen2.5-3B-Instruct	BLOCK with 5, 15, 45, or 135 queries/type	Held-out NONSENSE-*code*

Table 6: Curriculum and dataset-size configurations.

Experiment	Model	Training Data	Evaluation
Generator diversity	Qwen2.5-7B-Instruct	SCMs from GPT-5, GPT-5-mini, or Qwen2.5-7B-Instruct	Held-out GPT-5 NONSENSE-*code*

Table 7: Self-improvement experimental configuration.

A.6 Additional Experimental Analyses

The supplementary analyses are reported in dedicated appendices to avoid duplicating results. Appendix B compares absolute performance against larger, newer, and frontier models and gives the query-type breakdown in Table 9. Appendix C tests transfer of the training procedure to Llama 3.2 3B (Table 10). Appendix R.6 compares allocations of a fixed supervision budget between distinct SCMs and queries per SCM (Table 44).

A.7 Representative Dataset Examples

A.7.1 NONSENSE-*code*

```
def f_XQ(u_xq: str) -> str:
    if u_xq in ('aa', 'bb'):
        return 'XQ'
    return 'xq'
```

A.7.2 NONSENSE-*nl*

```
mevu is set as the returned output of f_MEVU.
The inputs to f_MEVU are u_mevu, BELU.

The Generation Rule for f_MEVU is: when u_mevu equals
"m", map BELU to "T" if BELU is True and to "F" if
BELU is False; when u_mevu equals "n", map BELU to
"Y" if BELU is True and to "N" if BELU is False.
```

A.7.3 MEDICAL-*code*

```
def f_infection_status(U_infection_exposure):
    if U_infection_exposure == 'none':
        return 'no_infection'
    elif U_infection_exposure == 'low':
        return 'colonized'
    else:
        return 'infected'
```

A.7.4 MEDICAL-*nl*

```
adaptive_status is set as the returned output of
f_adaptive_status. The input to f_adaptive_status is
U_vaccination_history.

The generation rule is: if U_vaccination_history equals
"none", return "naive"; if it equals "partial", return
"primed"; otherwise return "boosted".
```

A.7.5 Bayesian Networks from `bnlearn`

```
def f_Cancer(smoker, pollution, u_cancer):
    states = ['True', 'False']
    poll_index = 0 if pollution == 'low' else 1
```

Model	Small Training Set	Augmented Training Set	Evaluation
Qwen2.5-3B-Instruct	50 queries/type/size; 400 total	1,000 queries/type/size; 8,000 total	l_1–l_{10}; 8,000 held-out queries

Table 8: NIHSS augmentation configuration.

```
    smoke_index = 0 if smoker == 'True' else 1

    p_true = [
        [0.03, 0.001],
        [0.05, 0.02]
    ][poll_index][smoke_index]

    p_false = [
        [0.97, 0.999],
        [0.95, 0.98]
    ][poll_index][smoke_index]

    probs = [p_true, p_false]
    cum = 0.0

    for p, s in zip(probs, states):
        cum += p
        if u_cancer < cum:
            return s

    return states[-1]
```

A.7.6 Formalized GSM8K

```
def f_calories_remaining_for_dinner(
    calories_per_day,
    calories_at_breakfast,
    calories_at_lunch,
    calories_at_snack,
):
    return (
        calories_per_day
        - calories_at_breakfast
        - calories_at_lunch
        - calories_at_snack
    )
```

A.7.7 Partially Specified GSM8K

```
A cafe owner records 1 sandwich on Saturday and
2 on Sunday. Each sandwich uses an unknown 2 or
3 bread pieces; the factual record says 3.

Set the total number of bread pieces to 9 while
retaining the inferred facts. Return:

["num_of_pieces_of_bread_per_sandwich"]
```

A.7.8 NIH Stroke Scale

The NIHSS data begin from non-executable clinical scoring rules that are incrementally formalized into executable structural mechanisms. A representative rule and its formalized mechanism are given in Sec. A.3.4.

B Performance Anchoring Against Larger and Frontier Models

To contextualize the absolute performance reported in Sec. 5.1 and the magnitude of improvement from training, we compare our trained `Qwen2.5-3B-Instruct` model against several untrained reference models. We include: (i) its corresponding untrained `Qwen2.5-3B-Instruct` base model, (ii) the larger `Qwen2.5-7B-Instruct` model from the same model family, (iii) the newer-generation `Qwen3-4B-Instruct-2507`, and (iv) `GPT-5.5` as a frontier-model reference.

B.1 Evaluation Protocol and Query-Type Results

Trained model. We use the same `Qwen2.5-3B-Instruct` checkpoint as in Sec. 5.1: BLOCK training on NONSENSE-*code*, with 135 queries per type at each training size l_2, l_4, l_6, and l_8 (Appendix A.5). Here, "base" denotes the corresponding instruction-tuned checkpoint before our causal-reasoning training, not a separate pretrained-only model.

Evaluation sets and metric. All five models are evaluated on the same held-out NONSENSE-*code* and MEDICAL-*code* sets. Each set contains 6,000 queries: three topologies, 20 SCM IDs per topology, 10 incremental sizes, and five queries per type and size. Intervention and counterfactual queries therefore contribute 3,000 examples each. Table 9 reports Pass@1 (%) separately by query type and their average; Table 3 presents the average scores in the main paper. Correctness is determined using the executable verification procedure described in Sec. 4.3.

Scope of the comparison. These are absolute-performance reference points, not compute-matched comparisons: model scale, pretraining, and inference configuration are not controlled across model families. The results therefore quantify the observed performance gaps without isolating a compute-normalized training advantage.

Model	Avg. Pass@1	Counterfactual	Intervention
NONSENSE-*code*			
Qwen2.5-3B-Instruct, base	14.73	13.93	15.53
Qwen2.5-3B-Instruct, trained	39.18	43.47	34.90
Qwen2.5-7B-Instruct, base	22.48	22.33	22.63
Qwen3-4B-Instruct-2507, base	25.83	23.00	28.67
GPT-5.5	**44.73**	**44.43**	**45.03**
MEDICAL-*code*			
Qwen2.5-3B-Instruct, base	18.26	17.90	18.62
Qwen2.5-3B-Instruct, trained	31.07	31.34	30.79
Qwen2.5-7B-Instruct, base	28.90	29.24	28.55
Qwen3-4B-Instruct-2507, base	39.59	38.86	40.31
GPT-5.5	**60.88**	**60.41**	**61.34**

Table 9: Performance anchoring against larger, newer, and frontier models. Pass@1 (%) on the same held-out NONSENSE-*code* and MEDICAL-*code* sets.

Results. Training substantially improves the Qwen2.5-3B model over its untrained counterpart in both domains. The trained 3B model also outperforms the larger untrained Qwen2.5-7B model on both NONSENSE-*code* (39.18% vs. 22.48%) and MEDICAL-*code* (31.07% vs. 28.90%), showing that the improvement from causal training is not explained simply by model scale.

Second, relative to the newer Qwen3-4B model, the trained Qwen2.5-3B model performs better on the training-domain representation, NONSENSE-*code* (39.18% vs. 25.83%), but not on the unseen MEDICAL-*code* domain (31.07% vs. 39.59%). This comparison provides a reference point for the gains achieved under the current training curriculum and data scale, particularly under domain shift.

Finally, GPT-5.5 provides an estimate of frontier-model performance on these tasks. On NONSENSE-*code*, the trained 3B model reaches 39.18% Pass@1 compared with 44.73% for GPT-5.5. The gap is larger on the unseen MEDICAL-*code* domain, where the respective scores are 31.07% and 60.88%.

Domain	Model	Avg. Pass@1	Counterfactual	Intervention
NONSENSE-*code*	Base	10.57%	10.20%	10.93%
	Trained	**40.55%**	**43.20%**	**37.90%**
NONSENSE-*nl*	Base	11.40%	11.30%	11.50%
	Trained	**37.03%**	**38.80%**	**35.27%**
MEDICAL-*code*	Base	9.53%	8.93%	10.14%
	Trained	**26.50%**	**28.14%**	**24.86%**

Table 10: Generalization of the training procedure to Llama 3.2 3B. Training improves Pass@1 across in-distribution, representation-shift, and domain-shift evaluations.

C Generalization Across Model Families

Our main experiments use Qwen2.5 models: Qwen2.5-3B-Instruct in Secs. 5.1–5.2 and Qwen2.5-7B-Instruct in Sec. 5.3, demonstrating that the training procedure can be applied across model sizes. We additionally test whether the same procedure transfers to a different model family by training Llama 3.2 3B on NONSENSE-*code*.

We train on 180 SCM IDs spanning 1–10 endogenous variables, with five queries per SCM ID at each complexity level. Evaluation uses 20 held-out SCM IDs over the same range of graph sizes and query counts. We report performance on NONSENSE-*code*, NONSENSE-*nl*, and MEDICAL-*code*.

Training substantially improves Llama 3.2 3B across all three evaluation domains. Average Pass@1 increases from 10.57% to 40.55% on NONSENSE-*code*, from 11.40% to 37.03% on NONSENSE-*nl*, and from 9.53% to 26.50% on MEDICAL-*code*. These results show that the gains from our training procedure are not specific to the Qwen model family.

D 7B Generator Comparisons

This section reports the full generator-ablation results for Qwen2.5-7B-Instruct. The target model is fine-tuned on FORMAL NONSENSE SCMs generated by different sources: the target model itself, GPT-5-mini, and GPT-5. All models are evaluated on the same held-out GPT-5-generated benchmark.

Table 11: **Pass@1 performance on counterfactual queries with 5 training instances per SCM.** Qwen2.5-7B-Instruct is fine-tuned on **5** FORMAL NONSENSE instances per SCM, using 250 SCMs in total. Evaluation uses a held-out GPT-5-generated benchmark with 20 SCMs and 5 instances per SCM. Columns indicate the number of nodes.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.878	0.635	0.450	0.298	0.232	0.157	0.097	0.082	0.060	0.052
7B (self-generation)	0.878	0.660	**0.497**	0.357	0.265	0.213	0.162	0.113	**0.087**	0.072
GPT-5-mini	0.905	**0.720**	0.468	**0.360**	**0.295**	**0.218**	0.160	**0.115**	0.078	0.072
GPT-5	**0.925**	0.692	0.482	0.352	0.268	0.188	**0.178**	0.110	0.085	**0.075**

Table 12: **Pass@1 performance on intervention queries with 5 training instances per SCM.** Qwen2.5-7B-Instruct is fine-tuned on **5** FORMAL NONSENSE instances per SCM, using 250 SCMs in total. Evaluation uses a held-out GPT-5-generated benchmark with 20 SCMs and 5 instances per SCM. Columns indicate the number of nodes.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.898	0.625	0.438	0.308	0.228	0.185	0.130	0.108	0.065	0.052
7B (self-generation)	0.932	0.647	**0.475**	0.310	0.283	0.225	0.180	0.128	**0.108**	**0.078**
GPT-5-mini	**0.960**	0.655	0.472	**0.342**	**0.287**	**0.242**	0.175	**0.132**	0.105	0.077
GPT-5	0.958	**0.660**	0.460	0.300	0.280	0.223	**0.183**	0.112	0.093	0.062

Table 13: **Pass@1 performance on counterfactual queries with 15 training instances per SCM.** Qwen2.5-7B-Instruct is fine-tuned on **15** FORMAL NONSENSE instances per SCM, using 250 SCMs in total. Evaluation uses a held-out GPT-5-generated benchmark with 20 SCMs and 5 instances per SCM. Columns indicate the number of nodes.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.858	0.613	0.428	0.298	0.195	0.148	0.135	0.100	0.052	0.030
7B (self-generation)	0.900	0.653	0.533	0.347	0.295	0.197	0.175	0.132	**0.115**	0.082
GPT-5-mini	0.928	0.715	**0.540**	0.380	**0.318**	0.227	**0.180**	0.117	0.095	0.077
GPT-5	**0.945**	**0.728**	0.523	**0.395**	0.292	**0.233**	0.168	**0.147**	0.110	**0.092**

Table 14: **Pass@1 performance on intervention queries with 15 training instances per SCM.** Qwen2.5-7B-Instruct is fine-tuned on **15** FORMAL NONSENSE instances per SCM, using 250 SCMs in total. Evaluation uses a held-out GPT-5-generated benchmark with 20 SCMs and 5 instances per SCM. Columns indicate the number of nodes.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.895	0.625	0.415	0.293	0.257	0.148	0.130	0.092	0.060	0.043
7B (self-generation)	0.918	0.660	**0.508**	**0.352**	**0.322**	0.243	**0.195**	0.140	0.107	0.087
GPT-5-mini	**0.972**	0.675	0.480	0.343	0.293	**0.253**	0.165	**0.152**	0.098	**0.090**
GPT-5	0.958	**0.697**	0.462	0.350	0.277	0.228	0.185	0.140	**0.108**	0.075

D.1 Delta-Highlighted 7B Generator Comparisons

The following tables repeat the 7B generator-ablation results while additionally reporting absolute improvements over the base model in parentheses.

Table 15: **SCM generator ablation: Pass@1 performance on counterfactual queries with 5 training instances per SCM.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.88	0.64	0.45	0.30	0.23	0.16	0.10	0.08	0.06	0.05
7B (self-generation)	0.88 (+0.00)	0.66 (+0.03)	**0.50** (+0.05)	0.36 (+0.06)	0.27 (+0.03)	0.21 (+0.06)	0.16 (+0.07)	0.11 (+0.03)	**0.09** (+0.03)	0.07 (+0.02)
GPT-5-mini	0.91 (+0.03)	**0.72** (+0.09)	0.47 (+0.02)	**0.36** (+0.06)	**0.29** (+0.06)	**0.22** (+0.06)	0.16 (+0.06)	**0.12** (+0.03)	0.08 (+0.02)	0.07 (+0.02)
GPT-5	**0.93** (+0.05)	0.69 (+0.06)	0.48 (+0.03)	0.35 (+0.05)	0.27 (+0.04)	0.19 (+0.03)	**0.18** (+0.08)	0.11 (+0.03)	0.09 (+0.03)	**0.08** (+0.02)

Table 16: **SCM generator ablation: Pass@1 performance on intervention queries with 5 training instances per SCM.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.90	0.62	0.44	0.31	0.23	0.18	0.13	0.11	0.07	0.05
7B (self-generation)	0.93 (+0.03)	0.65 (+0.02)	**0.47** (+0.04)	0.31 (+0.00)	0.28 (+0.06)	0.23 (+0.04)	0.18 (+0.05)	0.13 (+0.02)	**0.11** (+0.04)	**0.08** (+0.03)
GPT-5-mini	**0.96** (+0.06)	0.66 (+0.03)	0.47 (+0.03)	**0.34** (+0.03)	**0.29** (+0.06)	**0.24** (+0.06)	0.18 (+0.04)	**0.13** (+0.02)	0.10 (+0.04)	0.08 (+0.03)
GPT-5	0.96 (+0.06)	**0.66** (+0.04)	0.46 (+0.02)	0.30 (-0.01)	0.28 (+0.05)	0.22 (+0.04)	**0.18** (+0.05)	0.11 (+0.00)	0.09 (+0.03)	0.06 (+0.01)

Table 17: **SCM generator ablation: Pass@1 performance on counterfactual queries with 15 training instances per SCM.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.86	0.61	0.43	0.30	0.20	0.15	0.14	0.10	0.05	0.03
7B (self-generation)	0.90 (+0.04)	0.65 (+0.04)	0.53 (+0.10)	0.35 (+0.05)	0.29 (+0.10)	0.20 (+0.05)	0.18 (+0.04)	0.13 (+0.03)	**0.12** (+0.06)	0.08 (+0.05)
GPT-5-mini	0.93 (+0.07)	0.72 (+0.10)	**0.54** (+0.11)	0.38 (+0.08)	**0.32** (+0.12)	0.23 (+0.08)	**0.18** (+0.04)	0.12 (+0.02)	0.10 (+0.04)	0.08 (+0.05)
GPT-5	**0.94** (+0.09)	**0.73** (+0.12)	0.52 (+0.10)	**0.40** (+0.10)	0.29 (+0.10)	**0.23** (+0.08)	0.17 (+0.03)	**0.15** (+0.05)	0.11 (+0.06)	**0.09** (+0.06)

Table 18: **SCM generator ablation: Pass@1 performance on intervention queries with 15 training instances per SCM.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.90	0.62	0.41	0.29	0.26	0.15	0.13	0.09	0.06	0.04
7B (self-generation)	0.92 (+0.02)	0.66 (+0.04)	**0.51** (+0.09)	**0.35** (+0.06)	**0.32** (+0.07)	0.24 (+0.09)	**0.20** (+0.07)	0.14 (+0.05)	0.11 (+0.05)	0.09 (+0.04)
GPT-5-mini	**0.97** (+0.08)	0.68 (+0.05)	0.48 (+0.07)	0.34 (+0.05)	0.29 (+0.04)	**0.25** (+0.10)	0.17 (+0.04)	**0.15** (+0.06)	0.10 (+0.04)	**0.09** (+0.05)
GPT-5	0.96 (+0.06)	**0.70** (+0.07)	0.46 (+0.05)	0.35 (+0.06)	0.28 (+0.02)	0.23 (+0.08)	0.18 (+0.06)	0.14 (+0.05)	**0.11** (+0.05)	0.07 (+0.03)

E Informal Representation Experiments

This section reports additional experiments evaluating transfer to informal SCM representations. Unless otherwise specified, all entries are pass@1 accuracies and columns indicate the number of nodes.

Table 19: **Pass@1 performance on INFORMAL-MEDICAL causal queries, averaged over selected prompt modes.**

Variant	1	2	3	4	5	6	7	8	9	10
Base	0.310	0.230	0.175	0.100	0.075	0.075	0.105	0.085	0.055	0.020
Train-135 (informal, non-sense)	0.375	0.305	0.285	0.225	0.170	0.145	0.120	0.140	0.090	0.085
Train-135 (formal, non-sense)	**0.535**	0.400	**0.470**	**0.305**	0.270	0.240	0.270	0.225	0.220	0.155
Train-135 (formal → informal, non-sense)	0.500	**0.410**	0.420	0.285	**0.395**	**0.305**	**0.340**	**0.310**	**0.270**	**0.205**

Table 20: **Pass@1 performance on INFORMAL-MEDICAL counterfactual queries.**

Variant	1	2	3	4	5	6	7	8	9	10
Base	0.310	0.190	0.140	0.070	0.080	0.080	0.090	0.020	0.040	0.020
Train-135 (informal, non-sense)	0.380	0.320	0.340	0.230	0.210	0.120	0.130	0.150	0.070	0.100
Train-135 (formal, non-sense)	**0.530**	0.450	0.480	**0.330**	0.340	0.260	0.290	0.240	0.240	0.170
Train-135 (formal → informal, non-sense)	0.480	**0.490**	**0.490**	0.320	**0.440**	**0.310**	**0.380**	**0.340**	**0.290**	**0.250**

Table 21: **Pass@1 performance on INFORMAL-MEDICAL intervention queries.**

Variant	1	2	3	4	5	6	7	8	9	10
Base	0.310	0.270	0.210	0.130	0.070	0.070	0.120	0.150	0.070	0.020
Train-135 (informal, non-sense)	0.370	0.290	0.230	0.220	0.130	0.170	0.110	0.130	0.110	0.070
Train-135 (formal, non-sense)	**0.540**	**0.350**	**0.460**	**0.280**	0.200	0.220	0.250	0.210	0.200	0.140
Train-135 (formal → informal, non-sense)	0.520	0.330	0.350	0.250	**0.350**	**0.300**	**0.300**	**0.280**	**0.250**	**0.160**

E.1 Delta-Highlighted INFORMAL-MEDICAL Out-of-Distribution Results

Table 22: **Pass@1 performance on INFORMAL-MEDICAL out-of-distribution causal queries, averaged over selected prompt modes.** All models use Qwen2.5-3B-Instruct. Train-135 (formal) and Train-135 (informal) are fine-tuned on 135 instances per SCM, using 180 SCMs in total. Train-135 (formal → informal) is trained sequentially on 135 formal and then 135 informal instances per SCM. Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.31	0.23	0.18	0.10	0.08	0.08	0.11	0.09	0.06	0.02
Train-135 (informal, non-sense)	0.38 (+0.07)	0.31 (+0.08)	0.29 (+0.11)	0.23 (+0.13)	0.17 (+0.10)	0.15 (+0.07)	0.12 (+0.02)	0.14 (+0.06)	0.09 (+0.04)	0.09 (+0.07)
Train-135 (formal, non-sense)	**0.54** (+0.23)	0.40 (+0.17)	**0.47** (+0.30)	**0.31** (+0.21)	0.27 (+0.20)	0.24 (+0.17)	0.27 (+0.17)	0.23 (+0.14)	0.22 (+0.17)	0.16 (+0.14)
Train-135 (formal → informal, non-sense)	0.50 (+0.19)	**0.41** (+0.18)	0.42 (+0.25)	0.29 (+0.19)	**0.40** (+0.32)	**0.31** (+0.23)	**0.34** (+0.24)	**0.31** (+0.23)	**0.27** (+0.22)	**0.21** (+0.19)

Table 23: **Pass@1 performance on INFORMAL-MEDICAL out-of-distribution counterfactual queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.31	0.19	0.14	0.07	0.08	0.08	0.09	0.02	0.04	0.02
Train-135 (informal, non-sense)	0.38 (+0.07)	0.32 (+0.13)	0.34 (+0.20)	0.23 (+0.16)	0.21 (+0.13)	0.12 (+0.04)	0.13 (+0.04)	0.15 (+0.13)	0.07 (+0.03)	0.10 (+0.08)
Train-135 (formal, non-sense)	**0.53** (+0.22)	0.45 (+0.26)	0.48 (+0.34)	**0.33** (+0.26)	0.34 (+0.26)	0.26 (+0.18)	0.29 (+0.20)	0.24 (+0.22)	0.24 (+0.20)	0.17 (+0.15)
Train-135 (formal → informal, non-sense)	0.48 (+0.17)	**0.49** (+0.30)	**0.49** (+0.35)	0.32 (+0.25)	**0.44** (+0.36)	**0.31** (+0.23)	**0.38** (+0.29)	**0.34** (+0.32)	**0.29** (+0.25)	**0.25** (+0.23)

Table 24: **Pass@1 performance on INFORMAL-MEDICAL out-of-distribution intervention queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.31	0.27	0.21	0.13	0.07	0.07	0.12	0.15	0.07	0.02
Train-135 (informal, non-sense)	0.37 (+0.06)	0.29 (+0.02)	0.23 (+0.02)	0.22 (+0.09)	0.13 (+0.06)	0.17 (+0.10)	0.11 (-0.01)	0.13 (-0.02)	0.11 (+0.04)	0.07 (+0.05)
Train-135 (formal, non-sense)	**0.54** (+0.23)	**0.35** (+0.08)	**0.46** (+0.25)	**0.28** (+0.15)	0.20 (+0.13)	0.22 (+0.15)	0.25 (+0.13)	0.21 (+0.06)	0.20 (+0.13)	0.14 (+0.12)
Train-135 (formal → informal, non-sense)	0.52 (+0.21)	0.33 (+0.06)	0.35 (+0.14)	0.25 (+0.12)	**0.35** (+0.28)	**0.30** (+0.23)	**0.30** (+0.18)	**0.28** (+0.13)	**0.25** (+0.18)	**0.16** (+0.14)

E.2 Informal-to-Informal Transfer on In-Domain Nonsensical SCMs: Inverted-Star Topology

Table 25: **Pass@1 performance on in-domain informal nonsensical SCMs with inverted-star topology, averaged over selected prompt modes.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.67	0.41	0.21	0.15	0.10	0.09	0.04	0.04	0.03	0.02
Train-135 (informal, non-sense)	**1.00** (+0.33)	0.63 (+0.22)	0.48 (+0.27)	0.35 (+0.20)	0.27 (+0.17)	0.21 (+0.12)	0.17 (+0.13)	0.16 (+0.12)	0.05 (+0.03)	0.07 (+0.05)
Train-135 (formal, non-sense)	0.93 (+0.26)	0.67 (+0.26)	0.59 (+0.39)	0.54 (+0.39)	0.43 (+0.33)	0.43 (+0.35)	0.40 (+0.36)	0.31 (+0.28)	0.20 (+0.17)	0.20 (+0.18)
Train-135 (formal → informal, non-sense)	0.98 (+0.31)	**0.75** (+0.34)	**0.71** (+0.51)	**0.68** (+0.53)	**0.59** (+0.49)	**0.59** (+0.51)	**0.55** (+0.51)	**0.52** (+0.49)	**0.38** (+0.35)	**0.41** (+0.39)

Table 26: **Pass@1 performance on in-domain informal nonsensical SCMs with inverted-star topology: counterfactual queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.72	0.42	0.18	0.15	0.09	0.06	0.01	0.04	0.03	0.01
Train-135 (informal, non-sense)	**1.00** (+0.28)	0.70 (+0.28)	0.40 (+0.22)	0.38 (+0.23)	0.22 (+0.13)	0.13 (+0.07)	0.20 (+0.19)	0.13 (+0.09)	0.03 (+0.00)	0.06 (+0.05)
Train-135 (formal, non-sense)	0.93 (+0.21)	0.71 (+0.29)	0.64 (+0.46)	0.55 (+0.40)	0.39 (+0.30)	0.46 (+0.40)	0.41 (+0.40)	0.29 (+0.25)	0.23 (+0.20)	0.18 (+0.17)
Train-135 (formal → informal, non-sense)	**1.00** (+0.28)	**0.77** (+0.35)	**0.70** (+0.52)	**0.71** (+0.56)	**0.59** (+0.50)	**0.65** (+0.59)	**0.55** (+0.54)	**0.57** (+0.53)	**0.36** (+0.33)	**0.41** (+0.40)

Table 27: **Pass@1 performance on in-domain informal nonsensical SCMs with inverted-star topology: intervention queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.62	0.40	0.23	0.15	0.11	0.11	0.07	0.03	0.02	0.03
Train-135 (informal, non-sense)	**0.99** (+0.37)	0.55 (+0.15)	0.55 (+0.32)	0.32 (+0.17)	0.32 (+0.21)	0.28 (+0.17)	0.14 (+0.07)	0.18 (+0.15)	0.07 (+0.05)	0.08 (+0.05)
Train-135 (formal, non-sense)	0.92 (+0.30)	0.63 (+0.23)	0.54 (+0.31)	0.52 (+0.37)	0.47 (+0.36)	0.40 (+0.29)	0.39 (+0.32)	0.33 (+0.30)	0.16 (+0.14)	0.22 (+0.19)
Train-135 (formal → informal, non-sense)	0.96 (+0.34)	**0.73** (+0.33)	**0.72** (+0.49)	**0.65** (+0.50)	**0.59** (+0.48)	**0.53** (+0.42)	**0.54** (+0.47)	**0.47** (+0.44)	**0.39** (+0.37)	**0.40** (+0.37)

E.3 Informal-to-Informal Transfer on In-Domain Nonsensical SCMs: Combined Topologies

Table 28: **Pass@1 performance on in-domain informal nonsensical SCMs, combined across all prompt modes.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.58	0.33	0.18	0.11	0.07	0.05	0.05	0.04	0.03	0.02
Train-135 (informal, non-sense)	**0.78** (+0.20)	0.51 (+0.19)	0.34 (+0.16)	0.24 (+0.13)	0.19 (+0.11)	0.15 (+0.09)	0.11 (+0.06)	0.10 (+0.06)	0.06 (+0.03)	0.07 (+0.05)
Train-135 (formal, non-sense)	0.68 (+0.10)	0.49 (+0.16)	0.38 (+0.19)	0.27 (+0.16)	0.21 (+0.14)	0.20 (+0.14)	0.18 (+0.13)	0.15 (+0.11)	0.12 (+0.09)	0.11 (+0.09)
Train-135 (formal → informal, non-sense)	0.73 (+0.14)	**0.55** (+0.22)	**0.41** (+0.23)	**0.32** (+0.21)	**0.31** (+0.23)	**0.27** (+0.22)	**0.25** (+0.20)	**0.22** (+0.18)	**0.18** (+0.15)	**0.18** (+0.15)

Table 29: **Pass@1 performance on in-domain informal nonsensical SCMs: counterfactual queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.59	0.34	0.18	0.10	0.07	0.05	0.04	0.02	0.02	0.02
Train-135 (informal, non-sense)	**0.77** (+0.18)	0.55 (+0.21)	0.34 (+0.17)	0.27 (+0.16)	0.19 (+0.13)	0.14 (+0.09)	0.12 (+0.08)	0.10 (+0.07)	0.06 (+0.04)	0.08 (+0.05)
Train-135 (formal, non-sense)	0.68 (+0.09)	0.52 (+0.18)	0.40 (+0.23)	0.29 (+0.18)	0.23 (+0.17)	0.20 (+0.16)	0.19 (+0.15)	0.15 (+0.13)	0.14 (+0.12)	0.12 (+0.09)
Train-135 (formal → informal, non-sense)	0.74 (+0.15)	**0.59** (+0.26)	**0.44** (+0.26)	**0.35** (+0.24)	**0.32** (+0.26)	**0.30** (+0.25)	**0.26** (+0.22)	**0.24** (+0.22)	**0.19** (+0.17)	**0.19** (+0.16)

Table 30: **Pass@1 performance on in-domain informal nonsensical SCMs: intervention queries.** Values in parentheses report absolute improvement over the base model.

Training data	1	2	3	4	5	6	7	8	9	10
Base	0.58	0.32	0.19	0.11	0.08	0.06	0.06	0.06	0.03	0.02
Train-135 (informal, non-sense)	**0.78** (+0.21)	0.48 (+0.16)	0.35 (+0.15)	0.22 (+0.11)	0.18 (+0.10)	0.15 (+0.09)	0.10 (+0.04)	0.09 (+0.04)	0.07 (+0.03)	0.06 (+0.04)
Train-135 (formal, non-sense)	0.69 (+0.12)	0.47 (+0.15)	0.35 (+0.16)	0.25 (+0.14)	0.19 (+0.11)	0.19 (+0.13)	0.17 (+0.11)	0.14 (+0.08)	0.10 (+0.06)	0.11 (+0.09)
Train-135 (formal → informal, non-sense)	0.72 (+0.14)	**0.50** (+0.18)	**0.38** (+0.19)	**0.29** (+0.18)	**0.29** (+0.21)	**0.24** (+0.18)	**0.23** (+0.17)	**0.20** (+0.14)	**0.17** (+0.13)	**0.16** (+0.14)

F Additional Benchmark and Ablation Results

This section provides additional benchmark results and diagnostic analyses used to support the main empirical claims.

F.1 Formalized External Benchmarks

We evaluate the NONSENSE-*code*-trained model on independently constructed SCMs derived from GSM8K and the ASIA, CANCER, and SURVEY Bayesian networks. The dataset configurations are given in Appendix A.5, with formalization prompts in Appendices A.3.5 and A.3.6.

Table 31: **Re-Imagine-derived benchmark.** Pass@1 performance on the formalized GSM8K evaluation set (configuration in Table 5). We compare the base Qwen2.5-3B-Instruct model with a version fine-tuned on 135 NONSENSE instances per SCM. The training set includes 180 SCMs for counterfactual and intervention queries; we additionally assess generalization to abduction and deduction tasks.

Model	Overall	Abd.	Ctf.	Ded.	Int.
Base	0.331	0.654	0.231	0.393	0.141
Train-135	**0.396**	**0.710**	**0.364**	**0.448**	**0.158**

Table 32: **Real-world causal graphs from bnlearn.** Pass@1 performance on bnlearn Bayesian networks converted to formal SCMs. We compare the base Qwen2.5-3B-Instruct model with a model fine-tuned on 135 NONSENSE queries per SCM.

	Overall		Counterfactual		Intervention	
Graph	Base	Train-135	Base	Train-135	Base	Train-135
ASIA	0.040	**0.155**	0.060	**0.190**	0.020	**0.120**
CANCER	0.065	**0.115**	0.070	**0.150**	0.060	**0.080**
SURVEY	0.010	**0.055**	0.020	**0.080**	0.000	**0.030**

F.2 Partially Specified GSM8K

This evaluation tests transfer beyond fully specified executable inputs. We present GSM8K-derived SCMs as unstructured natural-language problems with partially specified mechanisms; a representative input is provided in Appendix A.7.7. The evaluation configuration contains 262 SCMs and 500 intervention and counterfactual queries (Table 5). We compare the base `Qwen2.5-3B-Instruct` model with the same NONSENSE-*code*-trained model used in Sec. 5.1, without training on this evaluation distribution.

Table 33: **Re-Imagine-derived benchmark (Extremely Informal).** Pass@1 performance on the unstructured, partially specified GSM8K evaluation set. We compare the base Qwen2.5-3B-Instruct model with a version fine-tuned on 135 NONSENSE-*code* instances per SCM (TRAIN-135). Results are reported for average causal queries, counterfactual queries, and intervention queries.

Model	Avg. Causal	Ctf.	Int.
Base	0.1380	0.1218	0.1527
Train-135	**0.1940**	**0.1891**	**0.1985**

Average Pass@1 increases from 13.80% to 19.40%; counterfactual performance increases from 12.18% to 18.91%, and intervention performance from 15.27% to 19.85%. These gains support transfer to partially specified, non-executable inputs while leaving substantial headroom in absolute accuracy.

F.3 Pass@k and Majority-Vote Evaluation

Table 34 reports the additional multi-sample metrics for the external benchmarks. Corresponding NIHSS Pass@k results appear in Appendix R.2.

Table 34: **Pass@k and majority-vote performance on external benchmarks.** We compare the base Qwen2.5-3B-Instruct model with the model fine-tuned on 135 NONSENSE queries per SCM.

	Pass@k		Majority vote	
Dataset	Base	Train-135	Base	Train-135
ASIA	0.220	**0.235**	0.110	**0.165**
CANCER	0.300	**0.315**	0.100	**0.130**
SURVEY	0.135	**0.145**	0.025	**0.055**
ReImagine-derived	0.595	**0.623**	0.461	**0.469**

F.4 Failure Modes of One-Shot SCM Generation

As a preamble to our incremental setup, we characterize the dominant failure modes of one-shot SCM generation. Table 35 summarizes failures when prompting each model to generate a connected 20-node SCM in one prompt. Both models exhibit substantial failure rates, but the dominant failure mode differs: Qwen frequently produces disconnected graphs, whereas GPT-5 more often violates the requested node count.

Table 35: **Dominant error modes for one-shot generation of connected 20-node SCMs.** Results are measured over 100 trials. One-shot SCM generation becomes brittle at this scale.

Error type	**Qwen**	**GPT-5**
Graph is disconnected	75 / 100	3 / 100
Wrong number of nodes	3 / 100	44 / 100
Invalid syntax	14 / 100	3 / 100
Exogenous confounding	3 / 100	–
Correct	5 / 100	50 / 100

G Why Do We Want Causal Simulators?

Table 36 gives examples of domains where a causal simulator can support meaningful causal queries. The table illustrates the same core pattern across domains: a formal simulator encodes mechanisms, an informal representation describes those mechanisms in the format users typically encounter, SCM complexity can be scaled, and causal queries can be answered mechanically once the simulator is executable.

Table 36: **Formal and informal representations of SCM-based environments across domains.** The examples illustrate how simulator complexity and causal-query difficulty can scale together.

Setting	**Formal representation**	**Informal representation**	**Scaling SCM complexity**	**Example causal query**
Physics	Rigid-body simulator; U: initial conditions	Text, frames, or diagram	Add objects, forces, and longer horizons	If we set the initial speed higher, where does the object land?
Robot	3D manipulation simulator; U: friction and mass	RGB-D trace plus instruction	Add tools, contacts, and steps	Given a slip and failure, would the task succeed with a tighter grip?
Drug response	PK/PD code; U: patient type	Clinical note plus lab plot	Add compartments, interactions, and dosing rules	If we set the dose to high, what outcome follows?
Circuits	SPICE/HDL simulator; U: tolerances	Schematic plus waveforms	Add stages, feedback, and nonidealities	If we replace R_2 with 2kΩ, how does the output voltage change?
Math	Executable algebraic program; U: parameters	Equations or word problem	Add variables and compositions	If we force $x = x'$, what value(s) can y take?
Logic	Rule system or finite-state simulator; U: events	Natural-language rules or puzzles	Add rules, relations, and states	The alarm went off. Would it still go off if the sensor were off?

H LLM-Based SCM Generation

SCMs can also be generated without LLMs by sampling graphs and mechanisms from pre-defined priors. Here, we discuss why we explore LLM-based SCM generation and provide an initial empirical comparison against procedural SCM generators.

Why LLM-based SCM generation? We view LLM-based SCM generation as a valuable alternative to fixed-prior procedural methods for two main reasons: (1) reduced dependence on manually engineered priors, and (2) the ability to convert non-executable causal knowledge into executable SCMs.

Reduced dependence on manually engineered priors. To construct SCMs, a procedural generator samples from a manually specified prior over variables, edge probabilities, mechanisms, and exogenous noise distributions. An LLM also induces a prior, but one learned through broad pretraining and then conditioned on through prompts. It can therefore draw on pretrained knowledge to generate domain-specific SCMs, such as the mathematical and medical SCMs in our experiments, without requiring every component of the sampling distribution to be redesigned manually.

This reduced dependence on prior-distribution engineering connects to the motivation of recent work such as Absolute Zero [13]: as model capabilities improve, requiring humans to construct and enumerate the training distribution may itself become a limiting factor. This makes model-generated environments that are mechanically verifiable a valuable alternative to explore.

Conversion of non-executable causal knowledge into executable SCMs. A procedural generator can sample executable SCMs once the prior is defined. However, it does not by itself address the representation challenge identified in our introduction: causal knowledge is often available only as partial and non-executable descriptions. LLM-based construction allows this knowledge to be converted into executable SCMs and incorporated incrementally. This means that, as additional causal knowledge becomes available, new variables and mechanisms can be added step by step. A procedural generator could represent the same knowledge, but only after a human had incorporated it into the prior distribution.

Our current results therefore establish LLM-based generation with incremental construction and mechanical verification as one viable approach to SCM construction. A rigorous comparison against fixed-prior methods requires selecting and validating priors over variables, graph structures, mechanisms, and noise distributions, and conclusions may depend strongly on these choices and on the domain being evaluated. We therefore view a systematic comparison across different prior families as an important direction for future work.

H.1 Procedurally Generated SCM Baselines

To provide an initial empirical comparison against non-LLM data generation, we construct training datasets using procedural SCM generators and compare models trained on these datasets against models trained on our LLM-generated SCMs.

Data generation. A non-LLM SCM generator requires explicit specification of the prior from which each component of the SCM is sampled, including the graph structure, exogenous noise distributions, and structural mechanisms. We use the same connected random-DAG structure across both non-LLM settings, constructed incrementally by adding one variable at each step. Each exogenous variable is sampled independently from a Bernoulli distribution.

We evaluate two manually specified mechanism priors over NONSENSE variable names. Root variables copy their exogenous value, while each non-root variable applies either OR or XOR to its exogenous value and the values of all its parents. We generate a separate training dataset for each mechanism family, yielding **No-LLM: OR** and **No-LLM: XOR** baselines, and compare them against training on our LLM-generated NONSENSE-code SCMs.

Training setup. For each setting, we train Qwen2.5-3B-Instruct using the same number of SCMs and the same training setup.

Evaluation setup. We evaluate on held-out external datasets: GSM8K and the bnlearn Bayesian networks ASIA, CANCER, and SURVEY. GSM8K contains 270 independently constructed SCMs,

Evaluation Dataset	Base LLM	LLM	No-LLM: OR	No-LLM: XOR
GSM8K	25.38	**32.95**	30.68	31.06
ASIA	6.00	**23.00**	4.00	5.00
CANCER	8.00	**11.00**	5.00	10.00
SURVEY	4.00	**6.00**	5.00	2.00
Avg. over Datasets	10.84	**18.24**	11.17	12.02

Table 37: Counterfactual Pass@1 (%) when training on LLM-generated SCMs versus procedurally generated SCMs.

with 270 intervention and 264 counterfactual queries. ASIA, CANCER, and SURVEY each correspond to a single SCM and contain 200 queries: 100 intervention and 100 counterfactual queries. We report Pass@1 on counterfactual queries.

Results. Training on LLM-generated SCMs achieves the best performance on all four evaluation datasets and the highest average performance overall. Both non-LLM baselines improve over the base model on average, while performance also varies across the manually specified mechanism priors.

Discussion. These results support our motivation of reducing dependence on manually engineered priors. Procedurally generated SCMs provide useful training signal, but their performance varies with the specified priors over graph structure, exogenous noise, and mechanisms, as well as with how well these choices align with the evaluation distribution. In contrast, the LLM provides a broader learned prior without requiring each component of the SCM-generating distribution to be specified manually.

These experiments provide an initial empirical comparison rather than a comprehensive evaluation of LLM-based versus procedural SCM generation. A more systematic comparison across procedural priors, including different graph families, mechanism families, exogenous distributions, and domains, remains important future work.

I Discrete SCMs

We use discrete exogenous variables in our framework. This design choice serves two purposes: (1) discrete supports still require correct counterfactual reasoning, and (2) the discrete setting isolates causal reasoning from the additional mathematical demands of continuous probabilistic inference.

Discrete Supports Still Require Correct Counterfactual Reasoning. For any counterfactual query, the verifier first identifies the exogenous assignments that are compatible with the observed variables, applies the intervention under each of those assignments, and propagates its effects through the structural equations. The resulting set of possible outcomes forms the support of valid counterfactual answers, and the model is rewarded when its prediction matches any value in this support.

Importantly, the use of discrete variables does not make the counterfactual task trivial: producing a valid answer still requires the three steps of counterfactual reasoning:

1. **Abduction.** Reasoning from the observed variables to the exogenous assignments that could have generated them;
2. **Action.** Modifying the relevant structural equation through $\mathrm{do}(\cdot)$; and
3. **Prediction.** Propagating the exogenous assignments forward through the intervened SCM to obtain the corresponding counterfactual outcome.

Thus, in our discrete exogenous-noise setting, an LLM must correctly perform counterfactual reasoning to receive a reward. This aligns the training objective with our goal of improving causal reasoning.

The Discrete Setting Isolates Causal from Mathematical Reasoning. In a continuous setting, the same causal operations require additional probabilistic inference:

1. Abduction requires computing or approximating a posterior density over continuous exogenous variables.
2. Prediction requires propagating this density through the modified SCM and performing integration or density estimation.
3. Verification must account for the fact that exact equality to a single value is generally not an appropriate correctness criterion.

By using finite supports and exact executable verification, our framework allows us to test whether a model has learned the do-operator and the abduction–action–prediction procedure without conflating this with its ability to perform continuous probabilistic inference.

Future Work. Extending the framework to continuous-valued exogenous noise and stochastic mechanisms is an important direction for future work. We view this extension as distinct from the primary objective of the present work: designing a verifiable training environment for causal reasoning.

J Related Work Comparison

Table 38 compares our approach with prior SCM-based causal-reasoning work. The comparison highlights that our work focuses on incrementally constructing scalable executable SCMs and using them to improve causal reasoning through generated supervision.

Table 38: **Comparison of SCM-based approaches for causal reasoning.**

Approach	SCM type	How SCM rules are given	Improves reasoning	Abduction for counterfactuals	Scalable LLM-based benchmark	How SCM is created
Ours	Logic	Description	Yes	LLM simulates $p(u \mid \cdot)$	Yes, including complexity scaling	Step-by-step
[24]	Mathematical	Description	No	No	Yes, including mutation	One-shot
[34]	Logic	Description	Yes	No	No; static benchmark	Human-created
[19]	Spatial	Examples	No	No	No; static benchmark	Human-created
[6]	Mathematical	Description	No	LLM derives $p(u \mid \cdot)$	No; static benchmark	Human-created

K Causal Pattern Recognition versus Causal Mechanism Learning

A useful way to interpret the scope of our results is to distinguish causal-query answering from learning a causal model. In the structural causal model (SCM) view, causal knowledge is not only a joint distribution or a predictor over observed variables, but a collection of autonomous structural assignments $V_i := f_i(\mathrm{pa}_i, U_i)$, together with assumptions that license interventions and counterfactuals [15, 12, 45, 46, 47]. Observational, interventional, and counterfactual queries occupy distinct levels of the causal hierarchy: two models may agree on observational associations while disagreeing under interventions, and may further disagree on counterfactual queries even when some interventional distributions coincide [12, 46, 14]. Therefore, high supervised accuracy on a distribution of causal queries does not by itself imply that a neural model has identified the underlying causal graph, recovered the true modular mechanisms, or represented the data-generating process internally. The learner may instead exploit templates, lexical regularities, or other predictive shortcuts that are sufficient for the training and evaluation distribution but brittle under mechanism-level shifts [48, 49, 50].

Our simulator-based supervision should therefore be viewed as a controlled way to train and evaluate causal-query answering, rather than as proof that the resulting model has learned invariant causal mechanisms. A key advantage of our framework is that labels are generated by explicit executable SCMs and by mechanistic evaluation of observational, interventional, and counterfactual queries, rather than inferred from surface correlations in observational data. In addition, our use of synthetic nonsensical variables, progressively constructed mechanisms, and query curricula is intended to reduce reliance on common-knowledge priors and make the relevant causal computation more explicit. Nevertheless, the fine-tuning objective remains an input-output predictive loss, and standard work on causal generalization emphasizes that robust transfer typically requires invariances across environments and representations aligned with stable causal variables or mechanisms [51, 52, 53, 54, 50]. We therefore interpret our out-of-domain improvements as evidence that simulator-generated supervision can induce useful causal-reasoning behavior, but not as certification that the model's internal representation is itself an SCM.

Future work should more directly test whether the learned representations are mechanistic rather than merely predictive. One direction is to evaluate on held-out families of graphs, structural equations, noise models, and interventions that preserve superficial query form while changing the underlying causal semantics. Another is to train across multiple environments with explicit invariance, modularity, or mechanism-factorization constraints [51, 52, 53, 50]. Complementary analyses could inspect whether internal states align with SCM variables, use interchange interventions or causal-abstraction tests to verify whether those states play the expected causal role, and apply mediation-style analyses to determine whether putative mechanism representations are actually used in producing answers [55, 56]. Such evaluations would help distinguish models that recognize causal-looking patterns from models that encode transportable causal mechanisms, and would make the proposed simulator framework a testbed for causal representation learning in addition to causal-query supervision.

L Limitations of Self-Improvement

Our self-improvement experiment should be interpreted as a one-step bootstrapping result, not as evidence of unbounded recursive improvement. In our setting, the model is used to generate candidate SCMs, deterministic verification filters candidates for structural validity and executability, and the accepted SCMs are then executed to produce causal-query labels. This differs from unconstrained pseudo-labeling because the training labels are not the model's own free-form answers; they are mechanically computed from accepted simulators. Thus, for any accepted SCM, supervision is exact under the simulator semantics. This connects our method to prior self-training and reasoning-bootstrap approaches, where models improve by generating, filtering, and training on their own outputs [57, 40, 58], but with the important distinction that our verifier checks the generated environment rather than only the final answer.

Nevertheless, there is an inherent ceiling: the generator determines the distribution of SCMs from which supervision is obtained. Let G_θ denote the model-induced SCM generator and let $\mathcal{V}(M)$ be the verification predicate. Training data are drawn from the accepted distribution $p_{G_\theta}(M \mid \mathcal{V}(M) = 1)$, so the learner can only receive supervision for mechanisms, topologies, domains, and query structures that appear in this accepted support. Verification reduces syntactic and execution errors, but it does not by itself certify semantic richness, domain realism, mechanistic diversity, or coverage of rare causal structures. A weak generator may therefore produce valid but repetitive, shallow, or biased SCMs, creating a form of curriculum bottleneck. This is analogous to known limitations of self-training with noisy or biased pseudo-labels, where errors or biases can be reinforced over iterations [59, 60]. More broadly, work on synthetic-data feedback loops shows that repeatedly training on generated data can degrade quality or diversity when generated data replace or dominate fresh data [61, 62]; conversely, accumulating generated data together with real or externally grounded data can mitigate collapse [63].

Within the range studied in Sec. 5, we do not observe this saturation: performance continues to improve as additional simulator-generated supervision is introduced. The location of any eventual plateau is therefore an empirical property of the training setting and may depend on model capacity, the diversity and quality of the generated SCMs, and the alignment between the generated and evaluation distributions.

For this reason, our results establish that self-generated SCMs can be useful within the tested distribution and under deterministic verification, but they do not characterize the long-run fixed point of recursive self-improvement. Extending bootstrapping would require measuring the generator frontier across rounds: acceptance rate, number of unique mechanisms, graph-motif diversity, support sizes, query difficulty, and held-out performance on SCM families not generated by the current model. It would also require safeguards that expand rather than recycle the training distribution, such as mixing self-generated SCMs with externally generated or domain-grounded simulators, using independent validators, adding explicit diversity objectives, and periodically evaluating against out-of-generator benchmarks. Under such a view, self-improvement saturates when the accepted SCM distribution stops expanding or when additional accepted SCMs no longer provide new causal computations for the learner.

M Incremental Construction of Executable Structural Causal Models

This appendix provides implementation details for the incremental construction procedure used to generate executable structural causal models (SCMs). To reduce notational clutter, we identify each endogenous node with its mechanism name f_i, whose output is v_i.

M.1 SCM Specification

The user specifies the target number of endogenous variables n, a domain $\mathcal{D}$ such as logic-based symbolic mechanisms, and optionally a DAG topology G, such as a chain, tree, or star. These inputs are compiled into a prompt that asks the LLM to produce an initial executable SCM: exogenous samplers $\{U_i\}_{i=1}^n$, structural mechanisms $\{f_i\}_{i=1}^n$, and a driver function `run_once()`.

Each f_i takes its exogenous parent U_i and any endogenous parents explicitly as arguments, and returns a deterministic symbolic value. The resulting code is parsed and checked against the topology and SCM structural constraints described in Section M.4.

M.2 Planning

To grow an SCM by one node, we first plan a *symbolic* graph edit. The LLM is given a compact *semantic view* of the current SCM: the exogenous samplers $\mathcal{U} = \{U_1, \ldots, U_n\}$, the structural mechanisms $\mathcal{V} = \{f_1, \ldots, f_n\}$, the directed edge set $E \subseteq \mathcal{V} \times \mathcal{V}$, and a valid topological order π over $\mathcal{V}$. To convey meaning without exposing full low-level code, we also include short natural-language summaries, implemented as docstrings, for each U_i and f_i.

The planner outputs a modification decision specifying only: (i) the name of a new mechanism f_{n+1}, (ii) the name of its exogenous sampler U_{n+1}, (iii) its direct parents $Pa(f_{n+1}) \subseteq \mathcal{V}$, and (iv) its direct children $Ch(f_{n+1}) \subseteq \mathcal{V}$. No functional code is generated at this stage. The planner is instructed to maintain acyclicity, preserve standard SCM constraints, and obey the user-specified topology G. This symbolic plan is then passed to a second LLM call that performs the code update.

M.3 Execution

Given a planned edit, the executor implements it as a *localized patch*. To limit context, the LLM sees only: (i) full Python definitions for each direct child $f_j \in Ch(f_{n+1})$, since these functions must be updated, and (ii) docstrings for each direct parent $f_i \in Pa(f_{n+1})$, which provide semantic compatibility without exposing unnecessary code. No other SCM code is exposed.

The executor then: (i) defines the new exogenous sampler U_{n+1}, (ii) implements f_{n+1} as a deterministic function of $Pa(f_{n+1}) \cup \{U_{n+1}\}$, and (iii) updates each child $f_j \in Ch(f_{n+1})$ to take the output of f_{n+1} as an additional input, adjusting signatures and bodies as required by the topology.

M.4 Verification

After each edit, we apply deterministic post-generation checks to enforce the constraints the LLM was instructed to satisfy. We verify that the induced causal graph $G = (\mathcal{V}, E)$ is acyclic and connected, that each mechanism has exactly one corresponding exogenous parent, and that the number of endogenous variables matches the expected count. We additionally check compliance with the user-specified topology and require that the SCM executes without runtime errors in Python.

If any check fails, the edit is rejected and retried up to a user-specified limit. This verification loop ensures that the SCM remains a valid, executable causal simulator throughout construction.

Summary. Let $\mathcal{M}_0$ be an initial SCM with $|\mathcal{V}_0| = n_0$ endogenous mechanisms. At iteration t, we propose and verify an edit that yields an updated SCM $\mathcal{M}_t$ with

$$\mathcal{V}_t = \mathcal{V}_{t-1} \cup \{f_{n_0+t}\}.$$

Each update is accepted only if it passes all verification checks, producing a sequence $\{\mathcal{M}_t\}_{t=0}^T$ of progressively larger causal simulators. Algorithm 1 summarizes the procedure.

Algorithm 1 Incremental Construction of Executable Structural Causal Models

Require: Initial number of endogenous variables N_0, target number N, domain $\mathcal{D}$, optional DAG topology G, initialization retry limit K_0, update retry limit K
Ensure: Executable SCM $\mathcal{M}$ with N endogenous variables

Stage 0: SCM specification

for $r = 1$ to K_0 **do**
  Prompt LLM to generate initial SCM Python code: exogenous samplers $\mathcal{U}$, structural mechanisms $\mathcal{F}$, and driver function `run_once()`
  Verify acyclicity, connectedness, one exogenous parent per f_i, correct variable count, topology compliance, and Python execution
  if verification passes **then**
    Initialize $\mathcal{M}, \mathcal{U}, \mathcal{V}, E$, and $n \leftarrow N_0$
    break
  end if
end for
if initialization failed **then**
  Abort and report failure
end if

Stages 1–3: Incremental SCM updating

while $n < N$ **do**
  Construct semantic view of current SCM: $\mathcal{U}$, $\mathcal{V}$, edge set E, topological order π, and docstrings
  Prompt planner LLM to propose a symbolic edit: new mechanism f_{n+1}, new exogenous sampler U_{n+1}, parents $Pa(f_{n+1})$, and children $Ch(f_{n+1})$
  $accepted \leftarrow$ FALSE
  for $k = 1$ to K **do**
    Provide executor LLM with planner decision, full code of $f_j \in Ch(f_{n+1})$, and docstrings for $f_i \in Pa(f_{n+1})$
    Executor defines U_{n+1}, implements f_{n+1}, updates children $Ch(f_{n+1})$, and updates `run_once()`
    Verify the updated SCM
    if verification passes **then**
      Update $\mathcal{U} \leftarrow \mathcal{U} \cup \{U_{n+1}\}$, $\mathcal{V} \leftarrow \mathcal{V} \cup \{f_{n+1}\}$, and $n \leftarrow n + 1$
      $accepted \leftarrow$ TRUE
      break
    end if
  end for
  if $accepted =$ FALSE **then**
    Abort and report failure to extend SCM
  end if
end while
return Executable SCM $\mathcal{M} = (\mathcal{U}, \mathcal{V}, E, \texttt{run_once})$

N Generation Prompts

This appendix lists the prompts used in the incremental SCM construction procedure described in Appendix M.

Logic-Based Medical SCM Instantiation Prompt

```
SYSTEM_TEXT = You are a careful Python structural causal model
    (SCM) code generator.

USER_TEXT = You are generating a SMALL, EXECUTABLE, STOCHASTIC
    Structural Causal Model (SCM) in Python.

This SCM must be logic-based, symbolic, or mechanistic, based
    on a clinical, biological, or medical domain. DO NOT make
    the SCM mathematical.

Functions may return booleans, categorical strings, or symbolic
    states. They may use conditionals, for loops, and other
    logic statements.

HARD REQUIREMENTS:
- Allowed imports: random, typing. No others.
- Use docstrings ONLY (no '#' comments).
  Each docstring must include:
    - "Purpose:" Describe the logical or mechanistic rule and
        the variable's parents.
    - "Output:" Specify the exact symbolic/boolean values that
        the function may return.
- Define exogenous samplers named `U_*` (pure, no I/O). To
    support abduction:
  - Each `U_*` must be independent of all other exogenous
      variables.
  - There must be no exogenous confounding between variables.
- Define structural assignments named `f_*` (pure, no I/O):
  - Each `f_*` must take all parent variables, exogenous and
      structural, explicitly as arguments.
  - Each `f_*` must have exactly one exogenous parent.
  - The SCM must be acyclic.
  - Structural functions are deterministic given their inputs.
- Include a driver function:
    def run_once(seed: int | None = None) -> dict[str, object]:
  - If seed is not None, call random.seed(seed) inside the
      driver.
  - The driver must return a dict of output variables from f_*
      ONLY, not internal logic variables or outputs of U_*
      functions.

OUTPUT FORMAT, exact order:
1) All U_* exogenous sampler definitions.
2) All f_* structural mechanism definitions.
3) The run_once(seed) driver.

Return ONLY a single Python code block. No prose.

Task:
- Create an SCM with {num_nodes} exogenous samplers and
    {num_nodes} structural functions.
- The internal nodes (f_*) of the SCM must follow a {dag_type}
    topology. {DAG_DEFINITIONS[dag_type]}
```

Topology Definitions Used in Prompts

```
DAG_DEFINITIONS = {
    "chain": """
A chain topology is a single linear sequence of internal nodes.

Rules:
- Exactly one parent per node, except the first.
- Exactly one child per node, except the last.
- Shape: A -> B -> C -> ...
- If inserting a new node between X and Y:
    - Y must remove X as a parent.
    - Y must add the new node as its only parent.
    - The new node must take X as its parent.
    - The driver and signatures must reflect this reassignment.
- A new node may also be inserted at the start or end of the
    chain.
""",

    "star": """
A star topology has a single central parent, or hub, whose
    output flows to many children.

Rules:
- One hub node with many children.
- All children have exactly one parent, the hub.
- Children do not depend on one another.
- Shape: Hub -> {Child1, Child2, Child3, ...}
- New nodes can only be inserted as direct children of the hub.
""",

    "inverted_star": """
An inverted star topology is the reverse of a star: many
    parents converge into a single child, or collider.

Rules:
- A single sink node has multiple parents.
- All parents have exactly one child, the sink.
- The sink has no children.
- Parents do not depend on one another.
- Shape: {P1, P2, P3, ...} -> Child
- New nodes can only be inserted as direct parents of the sink
    node, with no direct parents themselves.
""",

    "tree": """
A tree topology has a single root and a branching structure
    with no merging.

Rules:
- One root node with no parents.
- Every other node has exactly one parent.
- Nodes may have multiple children.
- Shape: A -> {B, C}; B -> {D, E}
- A node can be inserted as the root, as a leaf, or between
    existing nodes.
    - If inserting a new node as the root R:
        - R must have no parents.
        - The previous root P must now have R as a parent.
    - If inserting a new node as a leaf L:
        - L must have no children.
    - If inserting a new node between parent P and child C:
        - C must remove P as its parent.
```

```
        - C must add the new node as its only parent.
        - The new node must take P as its parent.
""",

    "layered": """
A layered topology assigns every node to exactly one layer.

Rules:
- Multiple nodes can exist in one layer, but no edge can exist
    between nodes in the same layer.
- Allowed edges go only from layer k to layer k+1.
- No layer skipping, such as k -> k+2.
- No backward edges, such as k+1 -> k.

Inserting a new node:
- It may be placed in an existing layer or in a new layer.
- If inserted into an existing layer k:
    - The new node MUST NOT be a parent or child of any node in
        that layer.
    - Its parents must lie in layer k-1, if a previous layer
        exists.
    - Its children must lie in layer k+1, if a subsequent layer
        exists.
- If inserted into a new layer k:
    - The new node's direct children must lie in layer k+1.
    - The direct children must remove parents that are now two
        layers above.
    - Direct parents of the new node must lie in layer k-1.
""",

    "poly_tree": """
A polytree permits branching and merging while preserving
    acyclicity.

Rules:
- Multiple parents are allowed per node.
- Both branching, one parent to many children, and merging,
    many parents to one child, are allowed.
- When inserting a new node:
    - Children of the new node, if any, must update signatures
        and bodies accordingly.
    - The generator must preserve acyclicity.
"""
}
```

Planner Prompt: Deciding the Next Incremental Growth Step

```
You are planning the NEXT INCREMENTAL GROWTH of an existing
    logic-based, non-mathematical SCM, defined by the following
    functions:
- exogenous samplers (U_*) (external nodes)
- structural assignments (f_*) (internal nodes)

The ONLY permitted modification is adding a new internal node,
    which consists of:
- a new structural assignment f_*
- a new exogenous noise sampler U_* for the structural
    assignment

General constraints:
```

```
- The new node may be inserted anywhere in the graph as long as:
    - the graph remains acyclic.
    - all functions remain pure.
    - all modifications are compatible with topological
      execution.
    - no cycles are introduced.

Return STRICT JSON, with no prose, using this schema:
{
  "rationale": "one-sentence design intent",
  "new_function": "name of the new f_* function to create",
  "new_exo_noise": "name of the new U_* function to create",
  "direct_parents": ["f_*", ...],
  "direct_children": ["f_*", ...]
}

Important:
- direct_parents must include only f_* functions, never U_*
   functions.
- direct_children must include only f_* functions.
- The internal nodes (f_*) of the updated SCM must obey the
   strict {dag_type} topology.

SEMANTIC VIEW:
{semantic_view_json}

Return ONLY the strict JSON object above.
```

Executor Prompt: Implementing the Localized Update

```
You will IMPLEMENT a small update to a logic-based,
   non-mathematical structural causal model (SCM) in which ONE
   new internal structural function is added.

You are given:
1. A DECISION JSON with:
   - rationale
   - new_function: name of new f_*
   - new_exo_noise: name of corresponding U_*
   - direct_parents: list of internal parent nodes of
     new_function
   - direct_children: list of child nodes of new_function

2. CODE CONTEXT:
   - For each direct_child: full function, including signature,
     docstring, and body
   - For each direct_parent: docstring only, no code

3. DRIVER FUNCTION:
    def run_once(seed: int | None) -> dict[str, object]:

Your task:

(1) Define a new exogenous sampler U_new().
    - It must be pure and have no I/O.
    - It must return symbolic, categorical, or boolean states,
      not numeric math.
    - It must have a single docstring containing:
        Purpose: describe symbolic/mechanistic meaning
        Output: specify all symbolic values returned
```

```
    - It must be independent of all other U_* variables.

(2) Define the new structural function new_function:
    - Signature: one argument per direct_parent and the
      new_exo_noise.
    - Deterministic, logic-based, symbolic mechanism.
    - Single docstring containing Purpose and Output.
    - Must use ALL input arguments.

(3) Modify each direct_child:
    - Signature: add the new_function output as a new input in
      the signature and potentially remove other inputs
      depending on the specified topology.
    - Update docstring with Purpose and Output.
    - Update the body according to the newly specified inputs.
    - Preserve purity.

(4) Update the driver:
    - Insert call to U_new(), assigning it to a variable.
    - Insert call to new_function() with appropriate inputs,
      assigning output to a variable.
    - Update calls to direct_children to include the
      new_function output in the argument list, and
      potentially remove previously specified inputs depending
      on the topology.
    - Include outputs from new_function in the return statement.
    - Maintain topological order and purity.
    - Return ONLY outputs of structural functions, not
      exogenous variables.

Output format, Python code only:
    - Create the function for U_new.
    - Create the function for f_new.
    - Modify the direct_child functions to incorporate the
      output of f_new.
    - Modify the driver to make the new calls and amend the
      existing direct_child calls.

Constraints:
    - Allowed imports: random, typing.
    - Only docstrings; no '#' comments.
    - Non-mathematical: use symbolic, conditional, boolean, or
      categorical logic ONLY.
    - Do NOT emit unrelated functions.
```

O Evaluation Prompt Examples

This section gives representative prompts used to evaluate causal-query answering. The examples show how the executable SCM, query type, factual evidence, intervention, and required output format are presented to the model.

Evaluation Example: Real Causal Graph (Asia SCM)

```
def U_Cancer() -> float:
    choices = [i / 19 for i in range(20)]
    return random.choice(choices)

def U_Dyspnoea() -> float:
    choices = [i / 19 for i in range(20)]
    return random.choice(choices)

def U_Pollution() -> float:
    choices = [i / 19 for i in range(20)]
    return random.choice(choices)

def U_Smoker() -> float:
    choices = [i / 19 for i in range(20)]
    return random.choice(choices)

def U_Xray() -> float:
    choices = [i / 19 for i in range(20)]
    return random.choice(choices)

def f_Pollution(u_pollution: float) -> str:
    probs = [0.9, 0.1]; states = ['low', 'high']; cum=0.0
    for p, s in zip(probs, states):
        cum += p
        if u_pollution < cum: return s
    return states[-1]

def f_Smoker(u_smoker: float) -> str:
    probs = [0.3, 0.7]; states = ['True', 'False']; cum=0.0
    for p, s in zip(probs, states):
        cum += p
        if u_smoker < cum: return s
    return states[-1]

def f_Cancer(smoker: str, pollution: str, u_cancer: float) -> str:
    states = ['True', 'False']
    poll_index = 0 if pollution=='low' else 1
    smoke_index = 0 if smoker=='True' else 1
    p_true = [[0.03,0.001],[0.05,0.02]][poll_index][smoke_index]
    p_false=[[0.97,0.999],[0.95,0.98]][poll_index][smoke_index]
    probs = [p_true,p_false]; cum=0.0
    for p,s in zip(probs, states):
        cum+=p
        if u_cancer<cum: return s
    return states[-1]

def f_Dyspnoea(cancer: str, u_dyspnoea: float) -> str:
    states=['True','False']; probs=[0.65,0.35] if cancer=='True' else [0.3,0.7]
    cum=0.0
    for p,s in zip(probs,states):
        cum+=p
        if u_dyspnoea<cum: return s
    return states[-1]

def f_Xray(cancer: str, u_xray: float) -> str:
    states=['positive','negative']; probs=[0.9,0.1] if cancer=='True' else [0.2,0.8]
    cum=0.0
    for p,s in zip(probs,states):
        cum+=p
        if u_xray<cum: return s
    return states[-1]

def run_once(seed: int|None) -> dict[str,any]:
    if seed is not None: random.seed(seed)
    u_pollution=U_Pollution(); pollution=f_Pollution(u_pollution)
    u_smoker=U_Smoker(); smoker=f_Smoker(u_smoker)
    u_cancer=U_Cancer(); cancer=f_Cancer(smoker,pollution,u_cancer)
    u_dyspnoea=U_Dyspnoea(); dyspnoea=f_Dyspnoea(cancer,u_dyspnoea)
    u_xray=U_Xray(); xray=f_Xray(cancer,u_xray)
    return {'Pollution':pollution,'Smoker':smoker,'Cancer':cancer,'Dyspnoea':dyspnoea,'Xray':xray}
```

Evaluation Example: Informal Medical SCM

uv is set as the returned output of U_vaccination_history. The generation rule for U_vaccination_history is: generate a random number uniformly between 0 and 1; if it is less than 0.3, return "none"; else if it is less than 0.7, return "partial"; otherwise return "full".

ui is set as the returned output of U_infection_pressure. The generation rule for U_infection_pressure is: generate a random number uniformly between 0 and 1; if it is less than 0.5, return "low"; else if it is less than 0.85, return "moderate"; otherwise return "high".

ug is set as the returned output of U_genetic_immunity. The generation rule for U_genetic_immunity is: generate a random number uniformly between 0 and 1; if it is less than 0.25, return "weak"; else if it is less than 0.75, return "typical"; otherwise return "robust".

ubias is set as the returned output of U_infection_outcome_bias. The generation rule for U_infection_outcome_bias is: generate a random number uniformly between 0 and 1; if it is less than 0.2, return "attenuated"; else if it is less than 0.85, return "wildtype"; otherwise return "hypervirulent".

adaptive_status is set as the returned output of f_adaptive_status. The input to f_adaptive_status is U_vaccination_history. The generation rule is: if U_vaccination_history equals "none", return "naive"; if it equals "partial", return "primed"; otherwise return "boosted".

exposure is set as the returned output of f_exposure. The input is U_infection_pressure. The generation rule is: if U_infection_pressure equals "low", return "minimal"; if it equals "moderate", return "community"; otherwise return "close_contact".

innate_tone is set as the returned output of f_innate_tone. The input is U_genetic_immunity. The generation rule is: if U_genetic_immunity equals "weak", return "hypo"; if it equals "typical", return "normo"; otherwise return "hyper".

infection_outcome is set as the returned output of f_infection_outcome. The inputs are exposure, adaptive_status, innate_tone, and U_infection_outcome_bias. The generation rule is: first compute a risk score as the sum of three components derived from the inputs: base risk by exposure where "minimal" maps to 0, "community" maps to 1, and "close_contact" maps to 2; modification by adaptive_status where "naive" adds 2, "primed" adds 1, and "boosted" adds 0; modification by innate_tone where "hypo" adds 2, "normo" adds 1, and "hyper" adds 0. Then apply U_infection_outcome_bias: if it is "attenuated", subtract 1 but do not let the risk go below 0; if it is "wildtype", make no change; if it is "hypervirulent", add 1. Finally map the resulting risk score to the outcome: 0 yields "no_infection"; 1 or 2 yields "mild"; 3 or 4 yields "moderate"; any value 5 or greater yields "severe".

Prompt Example: Counterfactual Query

```
### System prompt
‘‘‘
You’re a helpful assistant solving problems based on user
    requests. The user provides a problem, possibly with
    instructions and a desired output format. You must first
    clearly outline your reasoning process enclosed within
    <think> reasoning process here </think>, followed
    immediately by the user’s requested output format.
‘‘‘

### User prompt
‘‘‘
```

```
Given this code:

from __future__ import annotations
import random
from typing import Optional
from typing import Dict
from typing import Any

def U_ZY() -> Any:
    pool = ['xo', 'mi', 'za', 'qe', 'tu']
    pick = random.choice(pool)
    if random.random() < 0.5:
        return pick
    else:
        return pick + random.choice(['', 'x'])

def f_ZY(u_zy: str) -> Any:
    s = u_zy
    if len(s) == 0:
        t = 'a'
    elif s[0] in ['x', 'q']:
        t = s + 'v'
    else:
        t = s + 'k'
    flag = 'x' in t and (t.endswith('v') or ('k' in t and (not
       t.endswith('v'))))
    if flag:
        return True
    else:
        for ch in ['m', 'z']:
            if ch in t and (not t.startswith('t')):
                return True
        return False

def run_once(seed: Optional[int]) -> Dict[str, bool]:
    if seed is not None:
        random.seed(seed)
    u_zy = U_ZY()
    zy = f_ZY(u_zy)
    return {'zy': zy}

Use the Python code above as the SCM definition.
Rules:
- Exogenous variables are named U_* and are sampled once per
    world.
- Endogenous (inner) variables are computed deterministically
    by their f_* equations given U_*.
- do(X=v) forces X=v and ignores f_X for that variable only;
    all other equations stay the same.
- Never resample randomness. For unknown U_*, treat them as
    free variables that can take any value allowed by the
    sampler code.

Query type: counterfactual
Scenario label: deterministic
Given:
- fixed_exogenous (known values): {}
- unknown_exogenous (missing names): ["U_ZY"]
- do (intervention): {"ZY":false}
- observed_endogenous (factual evidence, pre-do): {"ZY":true}

Task:
- Counterfactual = abduction + intervention + prediction.
```

```
  1) Abduction: infer unknown_exogenous assignment(s)
     consistent with observed_endogenous and fixed_exogenous.
  2) Intervention: apply do(...).
  3) Prediction: using the SAME abduced exogenous values (no
     resampling), compute required outputs under do(...).
  Important: observed_endogenous is only evidence for step (1);
     do NOT enforce it after the intervention.

  Return that unique output.

Required output keys (EXACT match; no extras): ["ZY"]

Answer format (STRICT):
- Your message may optionally start with a <think>...</think>
    block (reasoning).
- The LAST thing in your message must be the JSON object
    described below (so it can be parsed reliably).
- Output exactly ONE JSON object with exactly the required keys.
```

Prompt Example: Intervention Query

```
### System prompt
```
You're a helpful assistant solving problems based on user
    requests. The user provides a problem, possibly with
    instructions and a desired output format. You must first
    clearly outline your reasoning process enclosed within
    <think> reasoning process here </think>, followed
    immediately by the user's requested output format.
```

### User prompt
```
Given this code:

from __future__ import annotations
import math
import random
import statistics
from typing import Dict
from typing import List
from typing import Any
from typing import Literal
from typing import Optional

def U_ZY() -> Any:
    return random.choice(['ax', 'by', 'cz'])

def U_XQ() -> str:
    return random.choice(['aa', 'bb', 'cc'])

def U_VI() -> str:
    choices = ['ga', 'ho', 'ti']
    return random.choice(choices)

def U_TU() -> str:
    return random.choice(['ra', 'so', 'ti'])

def U_QE() -> str:
```
```

```
    if random.choice([True, False]):
        return 'qe'
    else:
        return 'qx'

def f_ZY(u_zy: str, XQ: str, VI: str, TU: str, QE: str) -> str:
    if XQ == 'XQ':
        if VI == 'VI':
            if TU == 'TU':
                if QE == 'QE':
                    if u_zy == 'ax':
                        return 'vok'
                    elif u_zy == 'by':
                        return 'pem'
                    else:
                        return 'nul'
                else:
                    return 'nul'
            else:
                return 'nul'
        else:
            return 'nul'
    else:
        return 'nul'

def f_XQ(u_xq: str) -> str:
    if u_xq in ('aa', 'bb'):
        return 'XQ'
    return 'xq'

def f_VI(u_vi: str) -> str:
    if u_vi in ('ga', 'ho'):
        return 'VI'
    return 'VX'

def f_TU(u_tu: str) -> str:
    if u_tu in ('ra', 'so'):
        return 'TU'
    return 'TU0'

def f_QE(u_qe: str) -> str:
    if u_qe == 'qe':
        return 'QE'
    else:
        return 'QX'

def run_once(seed: int | None) -> dict[str, str]:
    if seed is not None:
        random.seed(seed)
    u_xq = U_XQ()
    XQ = f_XQ(u_xq)
    u_vi = U_VI()
    VI = f_VI(u_vi)
    u_tu = U_TU()
    TU = f_TU(u_tu)
    u_qe = U_QE()
    QE = f_QE(u_qe)
    u_zy = U_ZY()
    zy = f_ZY(u_zy, XQ, VI, TU, QE)
    return {'xq': XQ, 'vi': VI, 'tu': TU, 'qe': QE, 'zy': zy}

Use the Python code above as the SCM definition.
Rules:
```

```
- Exogenous variables are named U_* and are sampled once per
    world.
- Endogenous (inner) variables are computed deterministically
    by their f_* equations given U_*.
- do(X=v) forces X=v and ignores f_X for that variable only;
    all other equations stay the same.
- Never resample randomness. For unknown U_*, treat them as
    free variables that can take any value allowed by the
    sampler code.

Query type: intervention
Scenario label: deterministic
Given:
- fixed_exogenous (known values):
    {"U_QE":"qe","U_TU":"so","U_VI":"ga","U_XQ":"cc","U_ZY":"by"}
- unknown_exogenous (missing names): []
- do (intervention): {"TU":"TU0"}

Task:
- Intervened world (apply do(...)).
  All exogenous values are fixed => the required outputs are
      uniquely determined under the intervention.
  Return that unique output.

Required output keys (EXACT match; no extras):
    ["QE","TU","VI","XQ","ZY"]

Answer format (STRICT):
- Your message may optionally start with a <think>...</think>
    block (reasoning).
- The LAST thing in your message must be the JSON object
    described below (so it can be parsed reliably).
- Output exactly ONE JSON object with exactly the required keys.
```

P Training Details

P.1 Training and Evaluation Pipeline

We use a staged pipeline that converts a library of structural causal models (SCMs) into prompts, collects baseline and training signals via inference, constructs a supervised fine-tuning (SFT) dataset by filtering model outputs, fine-tunes the model either in a single pass or as a curriculum [64], and finally re-evaluates on a fixed held-out test set. The pipeline is executed with a unified runner that selects an experimental variant, such as structure, curriculum, or data amount, and a run mode, such as full training, test-only re-run, or re-plotting. All generated artifacts are written into a per-project workspace, while the original SCM JSON store is treated as read-only.

Step 1: Data preparation: SCMs to prompts. For each domain and structure, the pipeline: (i) materializes SCMs into an executable form, (ii) deterministically selects train and test SCM IDs using fixed seeds, and (iii) builds prompt files for each task type, or prompt mode. For causal prompt modes, including deduction, abduction, intervention, and counterfactual, we first sample valid causal-query instantiations under deterministic constraints, such as intervention policy and counterfactual mode, and then format them into prompts. Train and test prompts are cached under dataset tags to ensure reproducibility and enable reuse across runs.

Step 2: Baseline inference before SFT. Before fine-tuning, the base model is evaluated on a fixed test set shared across experiments through fixed seeds and counts. This produces baseline metrics and plots. For efficiency, test prompts may be merged into a single inference job and split back into per-domain, per-structure, and per-prompt-mode result files.

Step 3: Training-signal collection before SFT. To obtain supervision, the base model is run on training prompts with multiple samples per instance. Outputs are stored as JSONL result files. If curriculum blocks are used, training results are organized by block to preserve the intended progression.

Step 4: SFT dataset construction by rejection sampling. The pipeline converts pre-SFT results into an SFT dataset by selecting one completion per prompt according to a selection strategy. The default strategy is `first_correct`. When curriculum blocks are enabled, the final SFT dataset is assembled in block order to preserve block boundaries.

Step 5: Fine-tuning. We support two regimes: (i) *single-shot* SFT, which trains once on the full block-ordered dataset; and (ii) *sequential curriculum* SFT, which iterates block-by-block, fine-tuning after each block and carrying the checkpoint forward. Optional LoRA fine-tuning is supported. When enabled, the pipeline automatically chooses micro-batch size, gradient accumulation, and learning rate based on dataset size and model family.

Step 6: Post-SFT inference and evaluation. The final checkpoint is evaluated on the same fixed test prompts as the baseline. The pipeline computes metrics and generates paper-ready plots comparing pre-SFT and post-SFT performance, with optional grouped comparisons across experimental conditions.

P.2 Fine-Tuning Hyperparameters

Table 39 reports the default SFT configuration used by the pipeline.

Table 39: **Default SFT configuration.**

Parameter	Default value
Epochs (`NUM_EPOCHS`)	2
Max sequence length (`MAX_SEQ_LENGTH`)	2048
Optimizer (`OPTIM`)	`adamw_torch_fused`
Warmup ratio (`WARMUP_RATIO`)	0.03
Weight decay (`WEIGHT_DECAY`)	0.0
Max gradient norm (`MAX_GRAD_NORM`)	1.0
bf16 (`BF16`)	1
tf32 (`TF32`)	1
Gradient checkpointing (`GRAD_CKPT`)	1
Dataloader workers (`DATALOADER_NUM_WORKERS`)	2
LoRA (`USE_LORA`)	0, off by default
Merge LoRA and save (`MERGE_LORA_AND_SAVE`)	1
4-bit quantization (`SFT_USE_4BIT`)	0, off by default
Dataloader shuffle (`NO_SHUFFLE`)	1 in single-shot training to preserve block order

Automatic hyperparameter selection. When `AUTO_SFT_HPARAMS=1`, the pipeline selects micro-batch size, gradient accumulation, and learning rate based on dataset size, bucketed by total examples, and model family, split into 3B and $\geq$7B settings. Tables 40 and 41 report the target effective global batch size used by the heuristic. For all experiments, we use 4A100.

Table 40: **Automatic SFT hyperparameter heuristic without LoRA.** `micro` is the per-device batch size, `ga` is gradient accumulation, and `global` is the target effective global batch size.

Dataset bucket	3B, no LoRA	7B, no LoRA
$\leq$7.5k, "5k"	micro=4, ga=2, global=16, lr=1×10^{-5}	micro=2, ga=4, global=16, lr=8×10^{-6}
$\leq$17.5k, "10k"	micro=4, ga=4, global=32, lr=2×10^{-5}	micro=2, ga=8, global=32, lr=1.5×10^{-5}
$>$17.5k, "25k"	micro=4, ga=8, global=64, lr=3×10^{-5}	micro=2, ga=16, global=64, lr=2×10^{-5}

Table 41: **Automatic SFT hyperparameter heuristic with LoRA.** `micro` is the per-device batch size, `ga` is gradient accumulation, and `global` is the target effective global batch size.

Dataset bucket	3B, LoRA	7B, LoRA
$\leq$7.5k, "5k"	micro=16, ga=1, global=32, lr=1×10^{-4}	micro=8, ga=2, global=32, lr=6×10^{-5}
$\leq$17.5k, "10k"	micro=16, ga=2, global=64, lr=1.5×10^{-4}	micro=8, ga=4, global=64, lr=8×10^{-5}
$>$17.5k, "25k"	micro=16, ga=4, global=128, lr=2×10^{-4}	micro=8, ga=8, global=128, lr=1×10^{-4}

Q Data Generation Procedure

This appendix describes the pipeline used to construct our datasets from stochastic structural causal models (SCMs). The pipeline consists of: (i) rehydrating SCM code from a serialized JSON state representation, (ii) sampling forward instantiations, or traces, from each SCM, (iii) generating causal-query instances for deduction, intervention, abduction, and counterfactual tasks with fully enumerated supports, and (iv) optionally assembling LLM-ready prompts.

Q.1 Support-Valued Semantics for Stochastic Queries

Our main experiments focus on deterministic query instances, where the verifier-induced answer support is a singleton. However, the same executable-SCM framework naturally extends to stochastic and partially observed settings. The key distinction in our approach is that we do not require the model to estimate calibrated probabilities over answers, since our goal is to study human-like causal reasoning: people typically do not answer causal queries by assigning explicit probabilities to all possible outcomes. Instead, each query is associated with a *support set* of answers that have nonzero probability under the SCM and the revealed information, and a model output is considered correct if it belongs to this support. We note that other formulations are possible; the support-set criterion is the one we use for abduction in this work, and it provides one natural way to extend causal-reasoning evaluations to stochastic settings.

Let

$$M = \langle \mathbf{U}, p(\mathbf{u}), \mathbf{V}, \mathbf{F} \rangle$$

be a discrete SCM. For a fixed exogenous assignment $\mathbf{u}$, let

$$\mathbf{v}(\mathbf{u}) \in \mathcal{V}$$

denote the full endogenous assignment obtained by executing the structural equations. If a query reveals only a subset of exogenous variables indexed by $\mathcal{I}$, with values $\bar{\mathbf{u}}_{\mathcal{I}}$, define the compatible exogenous contexts as

$$\Omega(\bar{\mathbf{u}}_{\mathcal{I}}) \coloneqq \{\mathbf{u} \in \mathrm{supp}(p) : \mathbf{u}_{\mathcal{I}} = \bar{\mathbf{u}}_{\mathcal{I}}\}.$$

Thus, stochasticity and missingness are both represented as multiple feasible exogenous completions. Smaller $\mathcal{I}$ corresponds to more missing exogenous information and therefore typically a larger answer support.

Deduction. A deduction query asks what endogenous assignment can occur given the revealed exogenous information. Its answer support is

$$\mathcal{A}^{\mathrm{ded}}(\bar{\mathbf{u}}_{\mathcal{I}}) \coloneqq \{\mathbf{v}(\mathbf{u}) : \mathbf{u} \in \Omega(\bar{\mathbf{u}}_{\mathcal{I}})\}.$$

In deterministic query instances, this support has size one. In stochastic or partially observed instances, several endogenous assignments may be valid.

Intervention. For an intervention $do(\mathbf{X} = \mathbf{x}^\star)$, let $\mathbf{v}_{do(\mathbf{X}=\mathbf{x}^\star)}(\mathbf{u})$ denote the endogenous assignment produced by replacing the structural equations for $\mathbf{X}$ with the externally assigned values $\mathbf{x}^\star$ while leaving all other mechanisms unchanged. The interventional answer support is

$$\mathcal{A}^{\mathrm{int}}_{do(\mathbf{X}=\mathbf{x}^\star)}(\bar{\mathbf{u}}_{\mathcal{I}}) \coloneqq \left\{\mathbf{v}_{do(\mathbf{X}=\mathbf{x}^\star)}(\mathbf{u}) : \mathbf{u} \in \Omega(\bar{\mathbf{u}}_{\mathcal{I}})\right\}.$$

Abduction. Abduction is the clearest stochastic case. Given endogenous evidence $\bar{\mathbf{v}}_{\mathcal{O}}$ over observed variables indexed by $\mathcal{O}$, the SCM induces a posterior distribution over exogenous contexts:

$$p(\mathbf{u} \mid \bar{\mathbf{u}}_{\mathcal{I}}, \bar{\mathbf{v}}_{\mathcal{O}}) \propto p(\mathbf{u})\, \mathbf{1}\{\mathbf{u}_{\mathcal{I}} = \bar{\mathbf{u}}_{\mathcal{I}}\}\, \mathbf{1}\{\mathbf{v}_{\mathcal{O}}(\mathbf{u}) = \bar{\mathbf{v}}_{\mathcal{O}}\}.$$

Our verifier does not require estimating this posterior distribution. It only uses its support:

$$\mathcal{A}^{\mathrm{abd}}(\bar{\mathbf{u}}_{\mathcal{I}}, \bar{\mathbf{v}}_{\mathcal{O}}) \coloneqq \{\mathbf{u} \in \Omega(\bar{\mathbf{u}}_{\mathcal{I}}) : \mathbf{v}_{\mathcal{O}}(\mathbf{u}) = \bar{\mathbf{v}}_{\mathcal{O}}\}.$$

An abductive answer $\hat{\mathbf{u}}$ is therefore correct whenever

$$\hat{\mathbf{u}} \in \mathcal{A}^{\mathrm{abd}}(\bar{\mathbf{u}}_{\mathcal{I}}, \bar{\mathbf{v}}_{\mathcal{O}}).$$

Equivalently, the model is rewarded for proposing any exogenous completion with nonzero posterior mass, not for recovering the posterior weights.

Counterfactuals. A counterfactual query combines abduction, action, and prediction. Given factual evidence $\bar{\mathbf{v}}_{\mathcal{O}}$ and an intervention $do(\mathbf{X} = \mathbf{x}')$, we first restrict to exogenous contexts consistent with the factual evidence and then evaluate the intervened SCM without resampling exogenous noise. The counterfactual support is

$$\mathcal{A}^{\mathrm{cf}}_{do(\mathbf{X}=\mathbf{x}')}(\bar{\mathbf{u}}_{\mathcal{I}}, \bar{\mathbf{v}}_{\mathcal{O}}) \coloneqq \left\{ \mathbf{v}_{do(\mathbf{X}=\mathbf{x}')}(\mathbf{u}) : \mathbf{u} \in \mathcal{A}^{\mathrm{abd}}(\bar{\mathbf{u}}_{\mathcal{I}}, \bar{\mathbf{v}}_{\mathcal{O}}) \right\}.$$

This preserves the usual counterfactual semantics: factual evidence is used only to abduce possible exogenous contexts, and the intervention is then evaluated under those same contexts.

Verification. For any query q, let $\mathcal{A}(q)$ denote the corresponding support set:

$$\mathcal{A}(q) \in \left\{ \mathcal{A}^{\mathrm{ded}}, \mathcal{A}^{\mathrm{int}}, \mathcal{A}^{\mathrm{abd}}, \mathcal{A}^{\mathrm{cf}} \right\}.$$

We score a model output $\hat{a}$ by support membership:

$$r(\hat{a}, q) = \mathbf{1}\{\hat{a} \in \mathcal{A}(q)\}.$$

Thus, exact-match verification in deterministic settings is a special case of support-membership verification where $|\mathcal{A}(q)| = 1$. In stochastic or partially observed settings, multiple answers may be valid, and the verifier rewards any answer that is feasible under the executable SCM.

Relation to probability estimation. The executable SCM also induces probability weights over the support. For example, a deduction or intervention query could define

$$P_q(a) = \sum_{\mathbf{u} \in \Omega_q} p(\mathbf{u} \mid q)\, \mathbf{1}\{g_q(\mathbf{u}) = a\},$$

where g_q denotes the relevant SCM execution map for the query and Ω_q is the set of exogenous contexts compatible with the query information. Our current framework uses only

$$\mathrm{supp}(P_q) = \{a : P_q(a) > 0\}$$

for verification. Learning calibrated probabilities over valid support elements is a natural extension, but is not required for the support-valued causal reasoning studied here.

Q.2 SCM Representation

Each SCM is stored in a JSON state file containing a list of SCM entries under a top-level `scms` field. An SCM entry contains an `scm_id` and a `history` list. Each element of `history` is a *bundle* corresponding to a model version `v`, including: (i) a set of exogenous sampler functions `U_*`, (ii) structural assignment functions `f_*` implementing deterministic structural equations conditional on exogenous noise, and (iii) a driver function `run_once(seed)` that samples a single world and returns outputs, either as a Python dictionary or as a dict-like object.

We use the convention that exogenous variables are sampled once per world, endogenous variables are computed via structural assignments, and interventions `do(X=x)` override only the equation for `X` while leaving all other structural equations unchanged.

Q.3 Stage 1: Rehydrating SCMs from JSON

We reconstruct standalone Python modules from each stored bundle using `rehydrate_scms_from_json.py`. For each SCM `scm_id` and each version `v` in an optional version range, the pipeline:

1. Writes an executable Python module `scm_{id}_v{v}.py` by emitting imports, function signatures, and function bodies from the bundle. A `from __future__ import annotations` header is included to avoid import-time failures due to unevaluated type annotations.
2. Optionally validates executability by importing the module and running `run_once(seed=0)`. Versions failing validation are skipped.
3. Writes a comment-stripped variant, `scm_{id}_v{v}_nocomments.py`, obtained by removing docstrings and #-style comments. This reduces prompt length and removes non-semantic text for downstream prompt construction.
4. Writes the bundle metadata itself to `bundle_v{v}.json`.

Parallelism and CPU hygiene. Rehydration and sampling are parallelized across SCMs using a process pool. The worker count is selected using cpuset-aware or cgroup-aware CPU detection, or scheduler environment variables such as SLURM and PBS, and capped by the number of SCMs. To prevent oversubscription when using multiprocessing, numerical backends are constrained by setting `OMP_NUM_THREADS`, `MKL_NUM_THREADS`, and related variables to 1 unless already set.

Q.4 Stage 1b: Sampling Forward Instantiations

For each validated SCM version, we generate a set of forward samples using `generate_instantiations` in `rehydrate_scms_from_json.py`. A single instantiation corresponds to one call to `run_once(seed)` and produces:

- the seed,
- the returned outputs, normalized to a dictionary when possible,
- a log of all values returned by exogenous samplers `U_*` during execution, and
- a log of all values returned by structural functions `f_*` during execution.

To obtain these traces, the pipeline monkey-patches `U_*` and `f_*` in the imported module with wrappers that record return values each time a function is called. Values are stored per function; if a function is called once, we store a scalar, and if it is called multiple times, we store a list.

Robust sampling under runtime errors. Some generated SCMs may fail for particular seeds, for example due to invalid casts or division by zero. Rather than aborting the entire SCM version, we iterate through increasing seeds and skip seeds that raise exceptions, collecting successful samples until a target count is reached. The number of attempts is bounded, by default proportional to the requested sample count with a hard cap, to avoid infinite loops on pathological models. If no seeds succeed, the version is marked as failed for instantiation generation. Instantiation files are written to `instantiations_v{v}.json`, and latest-version convenience copies are written to `instantiations.json` and `bundle.json`.

Q.5 Stage 2: Generating Causal-Query Datasets

From each SCM version, we construct supervised causal-query instances using `build_causal_query_instantiations.py`. Each instance is defined by: (i) a partial specification of the exogenous variables, with some fixed and others declared missing, (ii) optionally an intervention, (iii) optionally observed endogenous evidence, and (iv) a *support set* of all valid answers under the SCM semantics, subject to computational caps.

Domain inference. Before generating queries, the pipeline estimates discrete domains for exogenous and endogenous variables by forward sampling, using parameter `domain_samples`; the default shown in the code is 2000. These inferred domains are used to choose feasible subsets of missing variables and interventions, and to bound enumeration of supports.

Seeds and attempt schedule. For each SCM version, seeds are scanned in a contiguous range:

$$\texttt{seed} \in [\texttt{seed_start},\ \texttt{seed_start} + \texttt{seed_budget}),$$

with configurable `seed_start`, also settable by environment variable, and `seed_budget`. For each seed, we generate a factual world by sampling all exogenous variables and forward-computing the corresponding endogenous values. We then try a small number of randomized attempts per seed, controlled by `attempts_per_seed`, to construct a valid query meeting the desired constraints.

Missing-exogenous selection and feasibility caps. In the stochastic setting, we hide a subset of exogenous variables, controlled by `missing_exo_pct`, but enforce feasibility: the Cartesian product of missing-variable domain sizes is constrained by `max_combinations`. We also cap the final enumerated support size by `max_support_size`. Candidate missing variables with trivial domains, meaning size less than 2, are avoided.

Query types. We generate four query types, each stored as a JSON file with `meta` and `instances` fields.

1. **Deduction.** Given fixed exogenous values and a list of missing exogenous names, the model must predict the possible endogenous outputs in the factual, no-intervention world. The support is computed by enumerating all completions of missing exogenous assignments, within `max_combinations`, and forward-propagating through the SCM. We keep only instances where enumeration is exhaustive, `exhausted=True`, and non-empty. For the `deterministic` scenario label, we further require a unique output, with support size 1; for the `stochastic` scenario label, we require multiple outputs, with support size at least 2.
2. **Intervention.** This is analogous to deduction, but with a `do`-operator applied to a randomly selected subset of endogenous variables, controlled by `intervention_node_pct`. Intervention values are chosen to differ from the factual value when possible. We compute the enumerated support of outputs under the intervention. An optional constraint, `require_intervention_effect`, filters out degenerate deterministic interventions that do not change the output relative to the factual world.
3. **Abduction.** Given observed endogenous values from the factual world, the model must infer consistent assignments to missing exogenous variables. In the default formulation, all exogenous variables are treated as missing and the full endogenous assignment is provided as evidence. The support is the set of all exogenous completions consistent with the evidence, enumerated up to the same caps.
4. **Counterfactual.** Counterfactual queries implement the standard three-step semantics: (1) abduce exogenous values consistent with observed endogenous evidence, (2) apply an intervention `do`, and (3) predict outputs under the intervention using the same abduced exogenous values, with no resampling. We support two modes: a minimal-evidence mode where evidence is restricted to a small observed endogenous subset linked to a missing exogenous parent, selected using SCM graph metadata when available; and a full-evidence mode using all endogenous variables as observations. As with other query types, we require exhaustive enumeration and scenario-dependent support-size constraints. An optional `require_counterfactual_effect` filter removes deterministic counterfactuals with no effect.

Outputs. For each SCM `scm_id` and version `v`, causal-query instances are written to:
`scm_{id}/causal_queries/<query_type>/<scenario>/causal_queries_v{v}.json`,
and a convenience copy `causal_queries.json` is maintained for the latest version.

Q.6 Stage 3: Building Prompt Instances

Finally, we assemble LLM-ready prompt instances from the causal-query datasets using `build_causal_query_prompts.py`. Each prompt instance contains: (i) SCM code, preferably the comment-stripped module, or an unstructured natural-language SCM description if provided in metadata; and (ii) a task instruction block specifying the query type, given exogenous values, missing exogenous names, optional `do`, optional observed endogenous evidence, and strict output-formatting requirements, namely JSON with a fixed set of keys.

Each prompt also stores an `expected_output` object containing the enumerated support set and the required output keys. This enables automatic verification of model outputs.

Selection of complete SCMs and robustness. Prompt construction scans SCMs in the state file and selects the first N *complete* SCMs, where all required versions and causal-query files exist and meet minimum example counts. To mitigate attrition from invalid SCMs, the generation scripts optionally reserve a margin of extra SCMs, by default 20, during earlier processing. The prompt builder writes a sidecar file listing selected SCM IDs for reproducibility and, optionally, for aligning intervention instances with deduction instances.

Q.7 Summary

Overall, the dataset is generated by executable SCM reconstruction, robust forward sampling with seed-skipping for failures, exhaustive capped support enumeration for each query instance, and

optional prompt assembly for LLM evaluation. The pipeline records all hyperparameters, including seed ranges, support caps, missing-variable fractions, and intervention fractions, in per-file metadata to support exact reproduction.

R Additional Results

This section reports supplementary results for formal causal-query training, NIHSS data augmentation, pass@k evaluation, and additional ablations.

R.1 Training on Formal Causal Queries

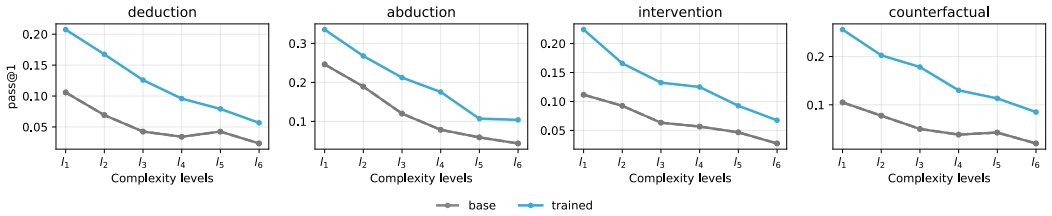


Figure 9: **Training on formal causal queries.** Pass@1 accuracy by causal query type for models trained with and without simulator-generated causal queries. Fine-tuning with causal-simulator data consistently improves performance over the base model.

R.2 Evaluation of Causal-Simulator Data Augmentation

We evaluate the value of transforming an existing informal SCM into executable form, automatically generating additional training data, and augmenting data in the original domain.

Experimental setup. We construct a controlled "original" dataset derived from the NIHSS dataset, consisting of 50 interventional and 50 counterfactual samples. We then convert this dataset into a formal SCM representation, following the procedure in Experiment 3, and generate an additional 1,000 interventional and counterfactual instances. We train two models: (i) a model trained only on the original dataset and (ii) a model trained on the original dataset augmented with simulator-generated data.

Results. Table 42 shows that adding causal-simulator data significantly improves performance in the unstructured domain, where ground-truth answers to causal queries would otherwise be unavailable.

Table 42: **Causal reasoning performance on the NIHSS dataset with and without causal-simulator data augmentation.** The augmented model uses 1,000 simulator-generated interventional and counterfactual instances, while the original model uses 50 of each. Columns indicate the number of nodes in the SCM.

Model	4	5	6	7	8	9
Original	0.239	0.230	0.261	0.255	0.213	0.191
Augmented	**0.423**	**0.344**	**0.398**	**0.351**	**0.349**	**0.378**

Table 43: **Pass@k performance on the NIHSS benchmark with and without causal-simulator data augmentation.** The augmented model, INT_CF_1000, uses 1,000 simulator-generated interventional and counterfactual instances. The base model uses 50 of each. Columns indicate the number of nodes in the SCM.

SCM size	1	2	3	4	5	6	7	8	9
Base	0.98	0.76	0.72	0.65	0.66	0.70	0.56	0.49	0.43
INT_CF_1000	**0.99** (+0.01)	**0.82** (+0.06)	**0.82** (+0.10)	**0.80** (+0.16)	**0.76** (+0.10)	**0.76** (+0.06)	**0.75** (+0.18)	**0.66** (+0.17)	**0.66** (+0.23)

R.3 All Causal Query Types

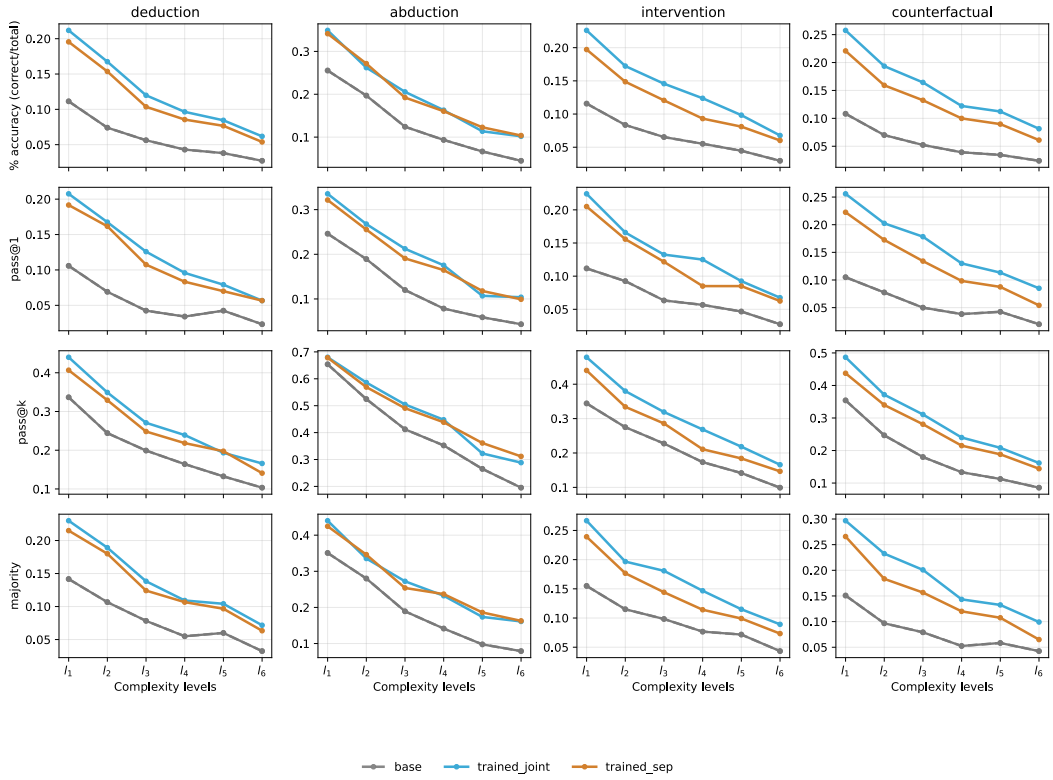


Figure 10: **Joint results across all causal query types.** Results for deduction, abduction, intervention, and counterfactual queries.

R.4 Data Amount Ablation

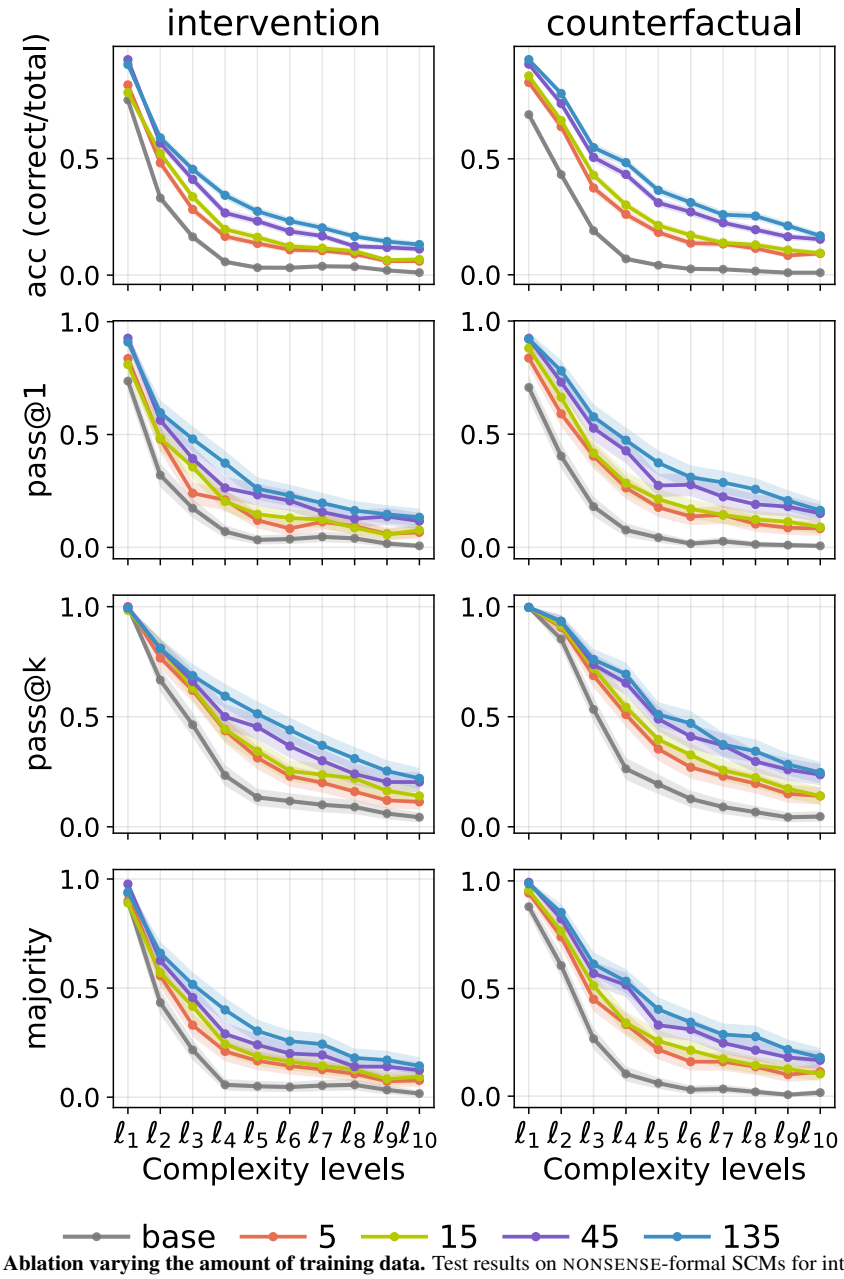


Figure 11: **Ablation varying the amount of training data.** Test results on NONSENSE-formal SCMs for intervention and counterfactual queries, averaged across structures. Models are trained jointly on counterfactual and intervention causal queries. We show levels 1-10.

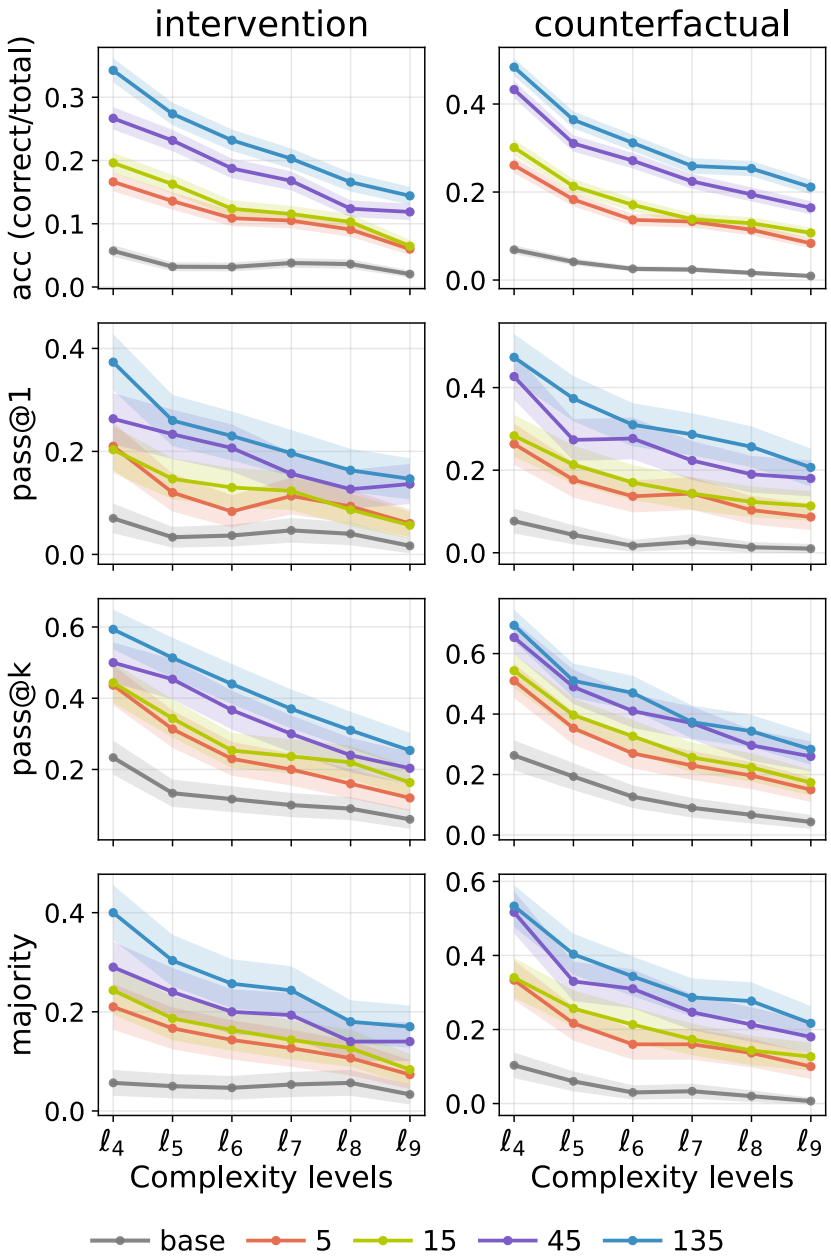


Figure 12: **Ablation varying the amount of training data.** Test results on NONSENSE-formal SCMs for intervention and counterfactual queries, averaged across structures. Models are trained jointly on counterfactual and intervention causal queries. We show levels 4-9.

R.5 Curriculum Ablation

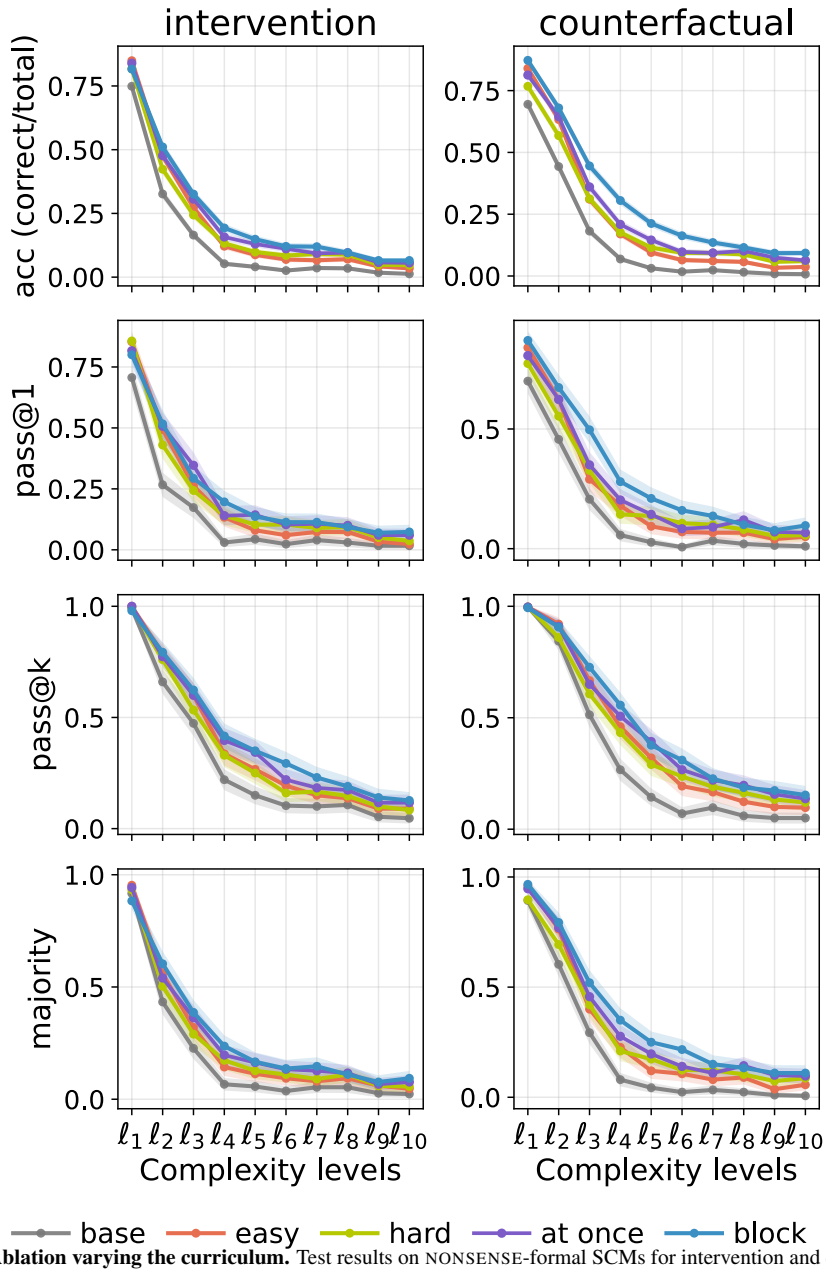


Figure 13: **Ablation varying the curriculum.** Test results on NONSENSE-formal SCMs for intervention and counterfactual queries, averaged across structures. Models are trained jointly on counterfactual and intervention causal queries. We show levels 1-10.

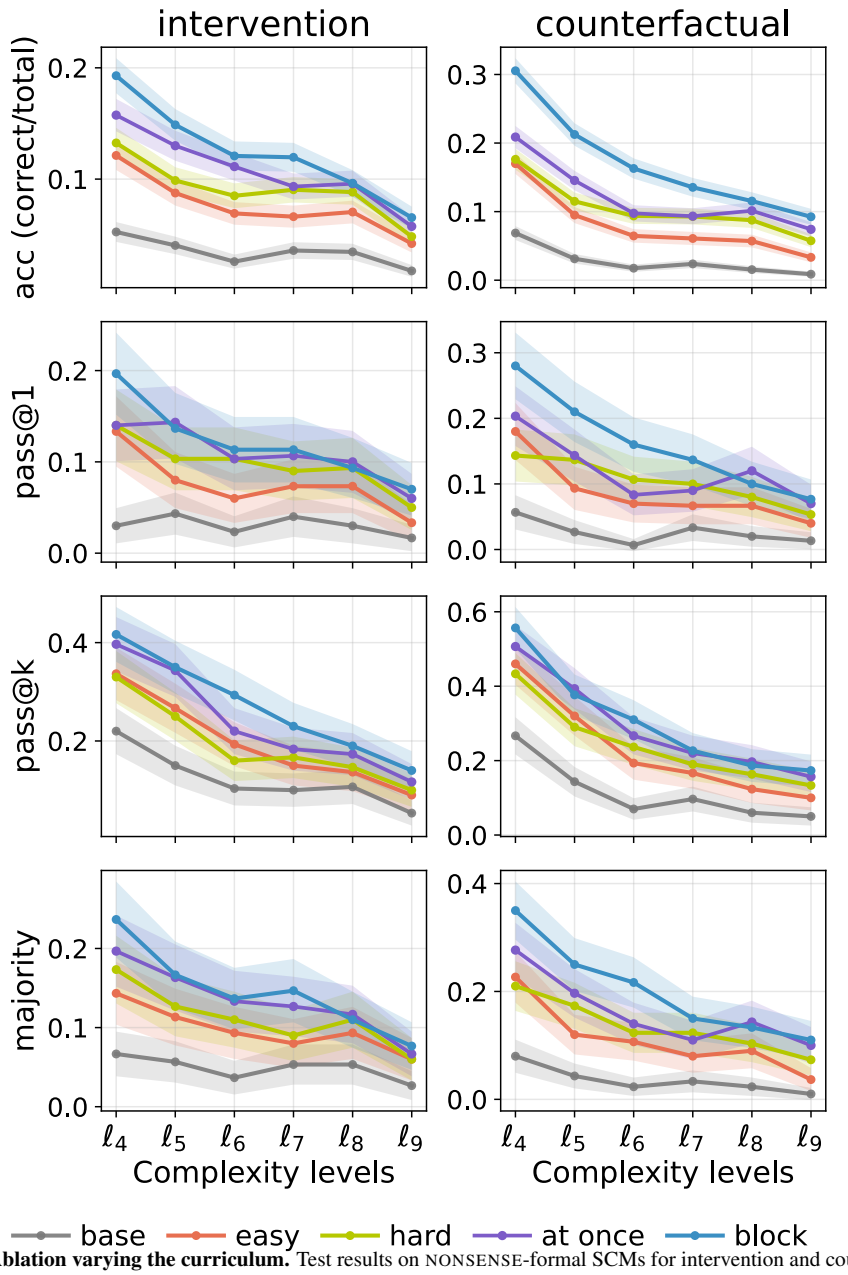


Figure 14: **Ablation varying the curriculum.** Test results on NONSENSE-formal SCMs for intervention and counterfactual queries, averaged across structures. Models are trained jointly on counterfactual and intervention causal queries. We show levels 4-9.

R.6 Number of SCMs vs. Queries per SCM

Our data-scaling experiments vary the total amount of generated supervision. We additionally ask whether, for a fixed query budget, it is preferable to increase the number of distinct SCMs or instead generate more queries from each SCM.

We train in the NONSENSE-*code* domain while varying the number of SCM IDs and the number of queries per (SCM ID, complexity level). Complexity ranges from 1 to 10 endogenous nodes. We compare three allocations with the same number of queries per complexity level: 180 SCM IDs with 5 queries each, 60 SCM IDs with 15 queries each, and 20 SCM IDs with 45 queries each. We evaluate on both held-out NONSENSE-*code* and MEDICAL-*code*.

Performance is highly consistent across the three allocations. Average Pass@1 varies by at most 0.40 percentage points on NONSENSE-*code* and 0.52 points on MEDICAL-*code*. Within the range

Evaluation	SCM IDs	Queries/SCM	Avg. Pass@1	Counterfactual	Intervention
NONSENSE-*code*	180	5	25.63	**28.23**	23.03
	60	15	25.72	27.63	23.80
	20	45	**26.03**	27.20	**24.87**
MEDICAL-*code*	180	5	24.98	24.52	**25.45**
	60	15	24.57	25.62	23.52
	20	45	**25.09**	**26.21**	23.97

Table 44: Ablation of the number of distinct SCMs versus the number of queries generated per SCM at a fixed query budget.

considered, we therefore find little sensitivity to whether a fixed supervision budget is allocated toward more distinct SCMs or more queries per SCM. A broader study of this breadth–depth trade-off remains an interesting direction for future work.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract/introduction claim a framework for scalable executable SCM construction, representation transfer, and simulator-based causal supervision; these are evaluated in Sec. 4 and Sec. 5, with scope limits discussed in the Discussion and App. K.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The Discussion notes limits of simulator-generated supervision, deterministic/mostly fully specified SCMs, and representation scope; further details appear in App. K and App. L.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not present new theoretical theorems or proofs; formal SCM definitions and query semantics are provided in Sec. 3.1 and App. Q.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper gives SCM construction algorithms, prompts, query generation, training pipeline, splits, and hyperparameters in App. M, App. N, App. O, App. P, and App. Q.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The paper does not provide open code or data access; reproduction details are documented in App. M, App. P, and App. Q.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Sec. 5 states the general setup, SCM splits, model, and query generation; App. P gives the training pipeline, optimizer, hyperparameters, and automatic hyperparameter rules.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Main and appendix line plots report 95% confidence intervals where applicable.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: App. P reports the use of 4 A100 GPUs.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work uses synthetic or benchmark-derived SCMs, does not involve human subjects, and is evaluated as a research method rather than deployed in a decision-making setting.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: The extend of our experiment remind academic with not possible harm at this stage.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release high-risk pretrained models, image generators, scraped datasets, or similarly misuse-sensitive assets.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [N/A]

    Justification: Not applicable, Our framework based on synthetic data generation, which does not fall in this category.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.

- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: No new dataset, codebase, or model artifact is released as an asset in the paper; generated SCM/query data are documented for reproducibility in App. Q.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing experiments or human-subject research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human-subject research, so IRB or equivalent approval is not applicable.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: LLMs are central to SCM generation, formalization/informalization, and self-generation experiments; usage is described in Sec. 4, Sec. 5, App. M, and App. N.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.