An Explanation In Support Of Neuro-Symbolic Language Models for Scaling Algorithmic Reasoning

Terry Tong [1] **Yu Feng** [1] **Surbhi Goel** [1] **Dan Roth** [1]

Abstract

Large language models can solve algorithmic problems either through direct natural language reasoning or by generating executable code delegated to an external solver. However, little theoretical progress has been made on explaining *why* code-based approaches consistently outperform natural language reasoning.

We introduce a three-arm framework that makes this comparison tractable by introducing an intermediary step—code generation with LLM-based execution—enabling pairwise theoretical analysis via Bayesian inference and information theory.

Empirically, we demonstrate on arithmetic, dynamic programming, and integer linear programming tasks that code execution achieves 78% accuracy versus 30% for code simulation and 21% for natural language reasoning across Deepseek and Gemma models ($p < 0.05$, Friedman test). Theoretically, we prove that code representations yield higher mutual information with the target algorithm, leading to at least 6% lower Bayes error than natural language.

These results inform the design of compositional AI systems, providing principled guidance on when to use tool-augmented versus monolithic reasoning for algorithmic tasks. Our framework offers a unified perspective on the tool-use versus direct-reasoning tradeoff.

1. Introduction

Large language models (LLMs) have demonstrated increasingly strong natural-language (NL) reasoning capabilities (). In parallel, LLM-based agentic systems advocate tool use, where LLMs invoke external solvers to support reasoning and execution (). Recent works (**??**) suggest that the **solver route**, translating problems into solver-executable representations and delegating execution, often outperforms the **direct route**, reasoning end-to-end in NL, on logic- and algorithmic-style complex reasoning tasks. However, for algorithmic reasoning alone, there is still no systematic analysis comparing the two routes, clarifying when and why LLMs perform better via the solver route versus the direct route.

A principled direct comparison is challenging since the routes operate over different representation spaces, i.e., NL traces versus solver-executable programs, and rely on different execution mechanisms, which prevents step-by-step alignment. Specifically, sample-complexity comparisons () are ill-posed here because the two routes learn fundamentally different objects, so there is no common formal target and metric to compare. Computational-complexity arguments () are also not a clean discriminator here because the two routes incur fundamentally different execution-

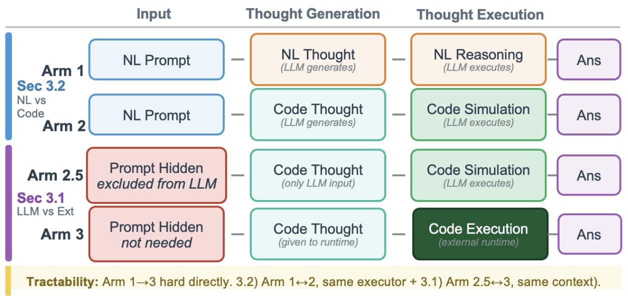
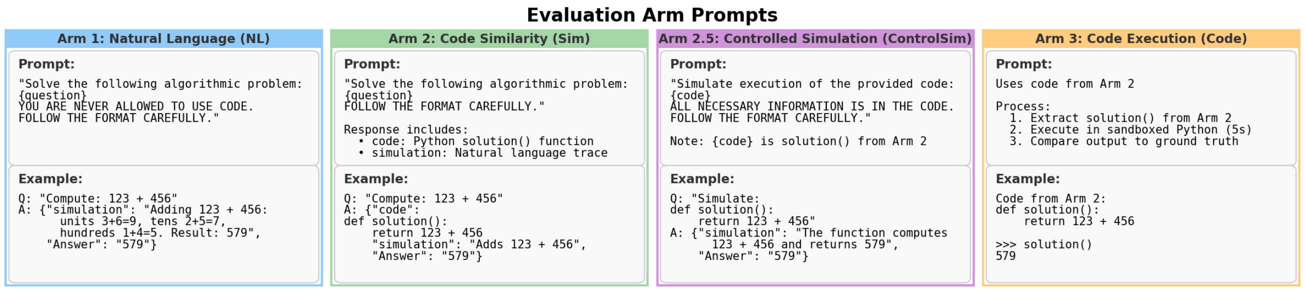


Figure 1. Bayesian Inference model showing the *three arms methodology* in Section 2. Given an algorithmic problem, we may split it into two steps (1) Translate (2) Execute. The (1) translation $\in \{\text{Code}, \text{NL}\}$. Then (2) execution $\in \{\text{LLM Reasoning}, \text{Solver Execution}\}$. We have three pairs, Arm 1: $\{\text{NL Gen}, \text{LLM Reasoning}\}$, Arm 2: $\{\text{Code Gen}, \text{LLM Reasoning}\}$, Arm 3: $\{\text{Code Gen}, \text{Solver Execution}\}$. Typically, the problem is tackled by comparing Arm 1 and Arm 3 in neuro-symbolic literature, which is intractable theoretically, and uncontrolled since multiple variables are changing. By introducing Arm 2, the problem becomes tractable. In the diagram, the *shaded* circles correspond to observed R.V. and *white* correspond to unobserved. The notation for R.V.s is correspondingly used in Section 2.

[*]Equal contribution [1]University of Pennsylvania. Correspondence to: Terry Tong <tongt1@seas.upenn.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).


Evaluation Arm Prompts
Arm 1: Natural Language (NL)
Arm 2: Code Similarity (Sim)
Arm 2.5: Controlled Simulation (ControlSim)
Arm 3: Code Execution (Code)

Arm 1: Natural Language (NL)

Prompt:

```
"Solve the following algorithmic problem:
{question}
YOU ARE NEVER ALLOWED TO USE CODE.
FOLLOW THE FORMAT CAREFULLY."
```

Example:

```
Q: "Compute: 123 + 456"
A: {"simulation": "Adding 123 + 456:
      units 3+6=9, tens 2+5=7,
      hundreds 1+4=5. Result: 579",
    "Answer": "579"}
```

Arm 2: Code Similarity (Sim)

Prompt:

```
"Solve the following algorithmic problem:
{question}
FOLLOW THE FORMAT CAREFULLY."

Response includes:
  • code: Python solution() function
  • simulation: Natural language trace
```

Example:

```
Q: "Compute: 123 + 456"
A: {"code":
def solution():
    return 123 + 456
    "simulation": "Adds 123 + 456",
    "Answer": "579"}
```

Arm 2.5: Controlled Simulation (ControlSim)

Prompt:

```
"Simulate execution of the provided code:
{code}
ALL NECESSARY INFORMATION IS IN THE CODE.
FOLLOW THE FORMAT CAREFULLY."

Note: {code} is solution() from Arm 2
```

Example:

```
Q: "Simulate:
def solution():
    return 123 + 456"
A: {"simulation": "The function computes
      123 + 456 and returns 579",
    "Answer": "579"}
```

Arm 3: Code Execution (Code)

Prompt:

```
Uses code from Arm 2

Process:
  1. Extract solution() from Arm 2
  2. Execute in sandboxed Python (5s)
  3. Compare output to ground truth
```

Example:

```
Code from Arm 2:
def solution():
    return 123 + 456

>>> solution()
579
```

Figure 2. Prompt templates for the three-arm evaluation framework. Arm 1 instructs the model to reason purely in natural language without code. Arm 2 instructs the model to generate code and simulate its execution. Arm 3 uses the same code generation prompt but executes the output in a Python runtime rather than simulating.

dependent costs. We therefore compare the two routes via statistical difficulty, using optimal achievable end-task Bayes risk ().

In this paper, we propose a three-route Bayesian inference framework that makes this comparison tractable by introducing an additional intermediate **simulation route**, where the model performs the same translation but simulates execution in NL, other than the **direct route** and **solver route**, and verbalizes the representation using Chain-of-Thought () as shown in Fig. 1. Using this framework, we characterize when and why algorithmic reasoning favors the solver route, and show that solver-based pipelines are generally easier for a broad class of tasks. [**Yu**: This claim is inaccurate for now.] This framework also enables a tractable theoretical comparison of the routes, showing that the simulation route outperforms the direct route [**Yu**: ....] Finally, we empirically demonstrate that the solver route substantially outperforms the simulation route, highlighting additional gains from reliable external execution.

Using our framework, we consistently observe a three-route ordering across algorithmic reasoning tasks: the solver route performs best, followed by the simulation route, and then the direct route. Moreover, the solver route's advantage widens as task difficulty increases. We evaluate our framework on CLRS30 (), NP-Hard-Eval (), and a custom suite of algorithmic problems with controllable difficulty (addition, multiplication, LCS, rod cutting, knapsack, and ILP variants: assignment, production, and partition) across a broad range of models, spanning weaker open-source LLMs (e.g., Mistral (), LLaMA (), Qwen ()) and stronger closed-source systems (e.g., OpenAI (), Gemini (), Claude ()). We find that, averaged over tasks and models, the solver route achieves XX% accuracy, outperforming the simulation route (XX%) and the direct route (XX%) with statistical significance (XX).

[**Yu**: theory part]

[**Yu**: talk about the recovery rate experiment to show why execution is better]

[**Yu**: Summarization of contributions:]

Theoretically, we first compare $\text{Arm } 1 < \text{Arm } 2$. Bayesian Inference shows us that the LLM implicitly does multi-class classification to the right algorithm. We utilize information theory to capture natural language and code under the same framework, forgoing using grammars or other mathematical frameworks that are intractable. We reduce the comparison of Bayes Error to that of comparing cross-entropy. A intermediate step using mutual information makes the proof interpretable: we prove that the mutual information between the CoT and the final answer is higher when conditioned on code representations over natural language representations. We variationally lower bound the mutual information using cross-entropy of a proposal distribution parameterized by a logistic regression. Since we only care about orderings, we subtract to overcome the intractability of estimating the differential entropy, reducing the comparison of mutual information to that of cross-entropy. The cross-entropy is measured empirically using logistic regression on TF-IDF features and Bert-base-uncased features, showing that code has lower cross-entropy than NL and achieves higher accuracy when classifying the correct algorithm. The difference is statistically significant (F-test, $p < 0.05$).

Then we compare $\text{Arm } 2 < \text{Arm } 3$. This difference is easily explained using a communication channel model of LLM forward-pass. We show that in the case where Arm 2 $>$ Arm 3, i.e. when the code generated is not executable or wrong, yet the LLM reasoning obtains the correct answer, that this occurs rarely. In other words, generally $\text{Arm } 3 > \text{Arm } 2$.

Piecing these results together, we show that $\text{Arm } 1 < \text{Arm } 2 < \text{Arm } 3$, verifying the hypothesis.

Understanding this problem is crucial as we move towards compositional AI systems rather than monolithic architectures.

Our main contributions are:

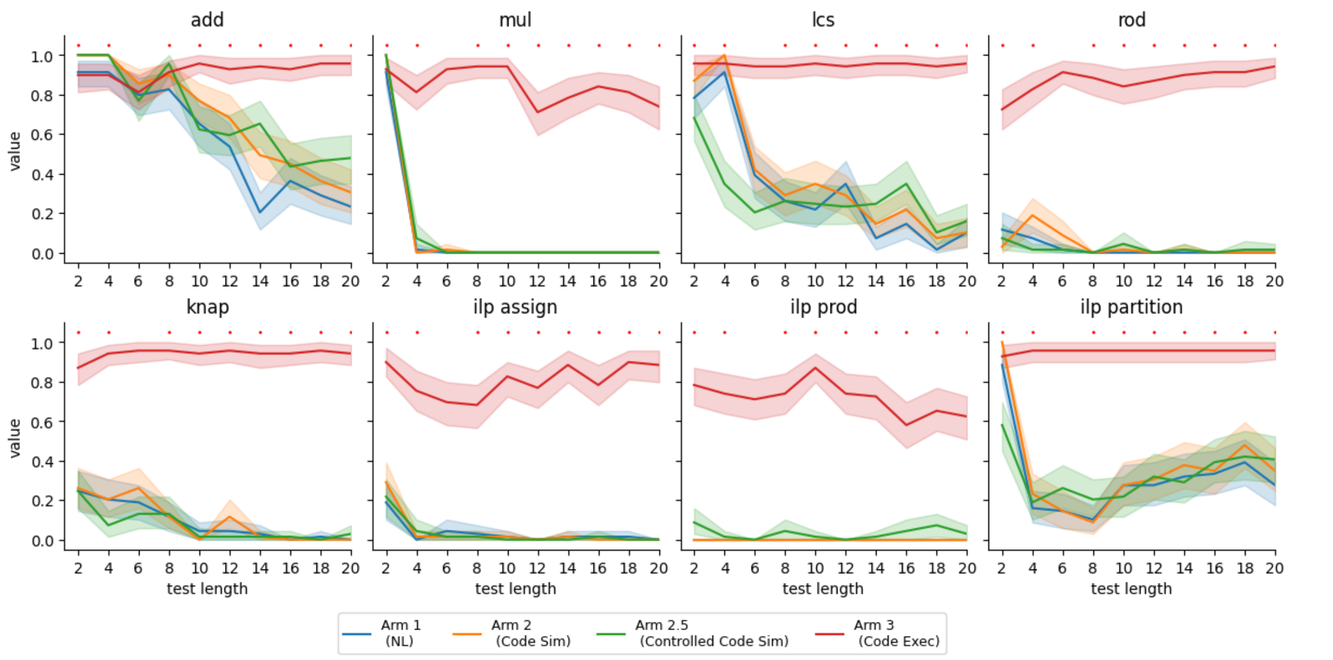


Figure 3. Accuracy scaling across task difficulty (τ) for Arithmetic, Dynamic Programming, and ILP tasks. Arm 3 (code execution) maintains high accuracy as problems get harder, while Arms 1 and 2 degrade. The widening gap demonstrates that solver execution becomes increasingly advantageous for challenging algorithmic problems.

1. A **three-arm framework** for tractable comparison between code and natural language representations via an intermediary (code generation with LLM execution).

2. **Empirical validation** demonstrating that code execution achieves 78% accuracy versus 21% for natural language reasoning across arithmetic, DP, and ILP tasks ($p < 0.05$).

3. A **theoretical explanation** based on Bayesian inference showing that code yields higher mutual information with target algorithms, leading to lower Bayes error.

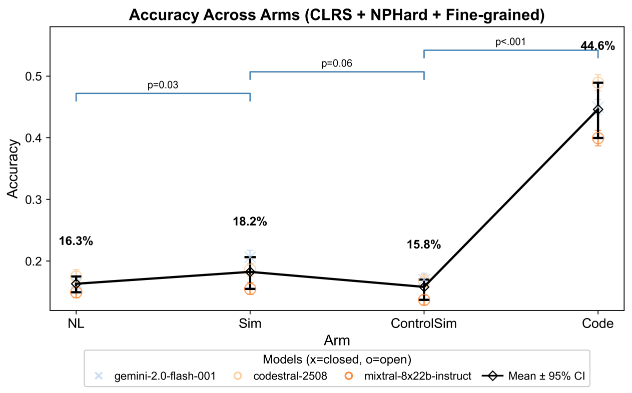


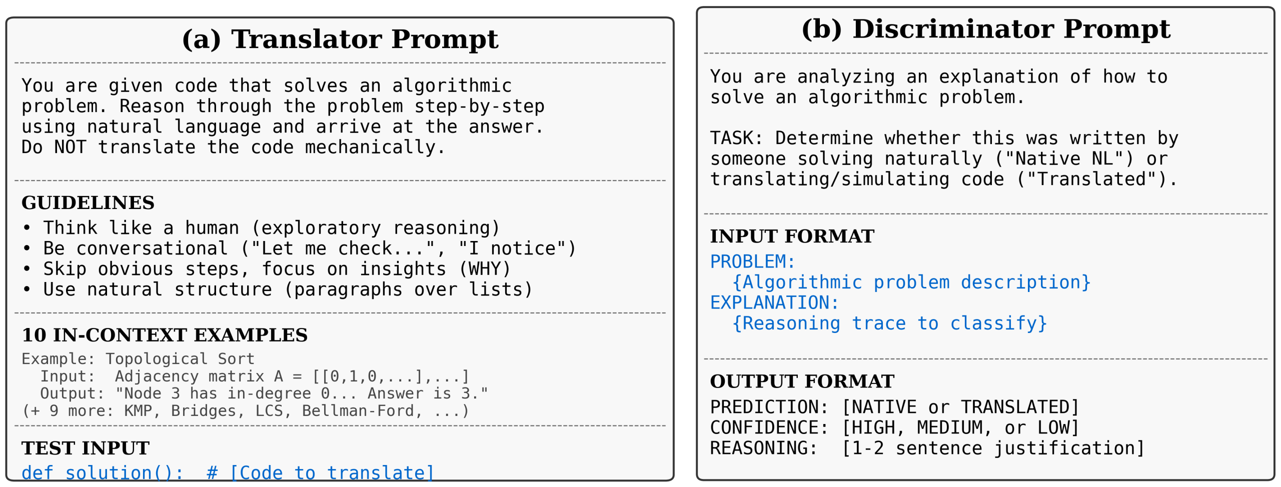
Figure 4. Average accuracy across arms for all models and tasks. Code execution (Arm 3) outperforms code simulation (Arm 2), which outperforms natural language reasoning (Arm 1) on CLRS$_{30}$ (n=500), NPHardEval (n=270), and Fine-Grained evaluation (n=270) benchmarks across 3 seeds. Statistical significance measured by Wilcoxon signed-rank test between adjacent arms. Error bars show bootstrapped Wilson confidence intervals at the model level.

2. Evaluation Framework

We formalize our central claim as $\mathrm{Acc}(\mathrm{Arm}\ 1) \leq \mathrm{Acc}(\mathrm{Arm}\ 2) < \mathrm{Acc}(\mathrm{Arm}\ 3)$. The following sections detail how we break down the problem and evaluate pairwise $\mathrm{Acc}(\mathrm{Arm}\ 1) \leq \mathrm{Acc}(\mathrm{Arm}\ 2)$ and then $\mathrm{Acc}(\mathrm{Arm}\ 2) \leq \mathrm{Acc}(\mathrm{Arm}\ 3)$.

2.1. Methodology

Arm 1 $\leq$ Arm 2 Setup. Similarly, given test input X_i corresponding to instance i, we prompt the LLM to reason about it using natural language in Arm 1, mapping $(X_i) \rightarrow Y_i^{(\mathrm{NL})}$. Likewise, Arm 2 can be written as using the same



(a) Translator Prompt

```
You are given code that solves an algorithmic
problem. Reason through the problem step-by-step
using natural language and arrive at the answer.
Do NOT translate the code mechanically.
```

GUIDELINES

```
• Think like a human (exploratory reasoning)
• Be conversational ("Let me check...", "I notice")
• Skip obvious steps, focus on insights (WHY)
• Use natural structure (paragraphs over lists)
```

10 IN-CONTEXT EXAMPLES

```
Example: Topological Sort
  Input:  Adjacency matrix A = [[0,1,0,...],...]
  Output: "Node 3 has in-degree 0... Answer is 3."
(+ 9 more: KMP, Bridges, LCS, Bellman-Ford, ...)
```

TEST INPUT

```
def solution():  # [Code to translate]
```

(b) Discriminator Prompt

```
You are analyzing an explanation of how to
solve an algorithmic problem.

TASK: Determine whether this was written by
someone solving naturally ("Native NL") or
translating/simulating code ("Translated").
```

INPUT FORMAT

```
PROBLEM:
  {Algorithmic problem description}
EXPLANATION:
  {Reasoning trace to classify}
```

OUTPUT FORMAT

```
PREDICTION: [NATIVE or TRANSLATED]
CONFIDENCE: [HIGH, MEDIUM, or LOW]
REASONING:  [1-2 sentence justification]
```

Figure 5. Recovery rate analysis: proportion of cases where LLM simulation (Arm 2) produces the correct answer despite incorrect code (Arm 3 failure). Recovery rates remain below 5% across all task families and models, confirming that solver execution dominates LLM simulation.

LLM to first $(X_i) \rightarrow C_i$ and then $(X_i, C_i) \rightarrow Y_i^{(\mathrm{Sim})}$. Example prompts are shown in Figure 2. We control the prompts to be as similar as possible, with the Arm 2 simply concatenating a section that says it should generate code and simulate it.

Arm 2 < Arm 3 Setup. Let the original problem statement corresponding to instance i be X_i, and let C_i be the corresoponding code generated for instance i. Let tuple (X_i, C_i) be fixed inputs in our experimental design. Let Arm 2 denote the output $Y_i^{(\mathrm{Sim})}$ which is mapped to by an LLM $(X_i, C_i) \rightarrow Y_i^{(\mathrm{Sim})}$, and let Arm 3 denote the output $Y_i^{(\mathrm{Exec})}$ mapped to by an external python runtime $(X_i, C_i) \rightarrow Y_i^{(\mathrm{Exec})}$ [1]. Example prompts are show in Figure 2.

Arm 1 $\leq$ Arm 2 < Arm 3 Setup. We observe bernoulli outcomes $(Y_i^{(\mathrm{NL})}, Y_i^{(\mathrm{Sim})}, Y_i^{(\mathrm{Exec})})$. Thus, we run Cochran's Q test across paired binary outcomes with the null $\mathrm{E}[Y_i^{(NL)}] = \mathrm{E}[Y_i^{(Sim)}] = \mathrm{E}[Y_i^{(Exec)}]$, testing whether at least one arm has a different marginal success accuracy.

Condition on failing to reject the null, following the framework, we break down the tests into Arm 1 and Arm 2 $(Y_i^{(\mathrm{NL})}, Y_i^{(\mathrm{Sim})})$, and Arm 2 and Arm 3 $(Y_i^{(\mathrm{Sim})}, Y_i^{(\mathrm{Exec})})$. For these paired bernoulli outcomes, we run the McNemar test under the null that $\Pr(Y^{(\mathrm{Sim})} = 1,\, Y^{(\mathrm{Exec})} = 0) = \Pr(Y^{(\mathrm{Sim})} = 0,\, Y^{(\mathrm{Exec})} = 1)$, that is, when pairs disagree, each one (Sim ¿ NL) and (NL ¡ Sim) occur equally as often. Similarly, this applies for the null for arm 2 and arm 3. To quantify effect size, we take paired accuracies $\Delta_{\mathrm{Exec-Sim}} = \mathrm{Acc(Exec)} - \mathrm{Acc(Sim)}$ and estimate its sampling distribution via cluster bootstrap resampling over instance pairs $(Y_i^{(\mathrm{Sim})}, Y_i^{(\mathrm{Exec})})$ corresponding to i. We do this for both Arm 1 vs Arm 2 and Arm 2 vs Arm 3.

Since we apply multiple statistical tests, we add Holm-Bonferoni corrections to control family wise error rate at $\alpha = 5\%$.

We run a GLMM with fixed effects and random effects, and observe how hardness interacts with the accuracy, and whether code or natural language helps modulate the performance.

2.2. Experiments

Data and Models. We use the CLRS 30 Benchmark (n=500), NPHardEval Benchmark (n=540), and a custom fine-grained evaluation suite (n=540), across three seeds. We define our own task suite—Arithmetic, Dynamic Programming, Integer Linear Programming (ILP)—to modulate hardness with parameter τ. For arithmetic, τ controls digit length; for DP, it controls table dimensionality; for ILP, it controls the constraint matrix size. We assume our algorithms are representative enough of the distribution. We select frontier models (Haiku 4.5, GPT-4o, Gemini 2.0 Flash) as well as open-source models (Mixtral, Codestral). Since we require structured output, we filter out models that give >50% JSON Parse Error, since this is indicative of instruction-following failures, rather than outright lack of coding fidelity.

Prompting to generate Code and Reasoning Traces. We

[1] X_i is ignored by executor, but we include it for notational symmetry

prompt the LLM in Arm 1 to never use any code in its reasoning, and to give a structured output of the rationale and answer to the algorithmic problem. Similarly for Arm 2, we prompt the LLM to use code in its reasoning, generating a structured output that contains a piece of code, and an attempt at simulating the execution of that piece of code in natural language, followed by a final answer. For Arm 3, we take the generated function (no prompt), and execute it in a python3 runtime. Models have access to native python packages, and numpy, pandas, scipy, PuLP, and pytorch. Figure 2 illustrates the prompt templates used across all three arms.

Arm 1 $\leq$ Arm 2 $<$ Arm 3 Results. Across 6 different types of models, and 44 algorithmic tasks spanning the CLRS 30 benchmark, NPHardEval benchmark and a custom suite of evals (Appendix), we strongly reject the null hypothesis, observing (p ¡ 0.001) and that the paired accuracy gap is strictly large and positive, excluding zero in 95% bootstrap confidence interval. Observing the distribution, the result is not driven by outliers, given the tightness of the distribution, indicating advantages over instance pairs and not just the average. This means that LLM execution via natural language reasoning of a piece of code has statistically significantly worse performances than code execution under the representative benchmarks and suites of task.

we run 30,000+ samples total and reject the null (p=0.04). However, the results are inconclusive since, under the cluster bootstrap, we see that the distribution's confidence interval overlaps with 0.0, meaning that natural language reasoning and code simulation are approximately equal, with code simulation occasionally being better (not strict inequality). For the purposes of our framework, instrict inequality is ok.

Conclusive across both open and closed models.

Advantages of Arm 3 emerge as tasks get harder. Figure 3

3. Evaluating Translated NL and NL Distributional Similarity.

One key hypothesis is that natural language reasoning follows a deeper algorithmic procedure encoded in its representations. If this is the case, it would make sense to surface the code and send it to an executor. We test whether natural language reasoning generated by information contained in the code alone can be distributionally similar to natural language reasoning generated from the prompt alone in Arm 1. Then, we test whether they are functionally similar.

3.1. Distributional Similarity between Translated NL and original NL

We show that the distribution of traces produced directly from the reasoning model can be approximated by post-processing the code with a fixed function across tasks and models. For each task instance x drawn from a held-out test distribution over 21 different algorithmic problems in CLRS, we consider two conditional distributions: Arm 1: $p_{NL}(\cdot \mid x)$, original NL traces generated by an LLM (Gemini 2.0 Flash in our case) and Arm 2 Translated: $p_{Translated}(\cdot \mid x)$, NL traces obtained by first generating code by an LLM, then translating that code into natural language using a fixed translator.

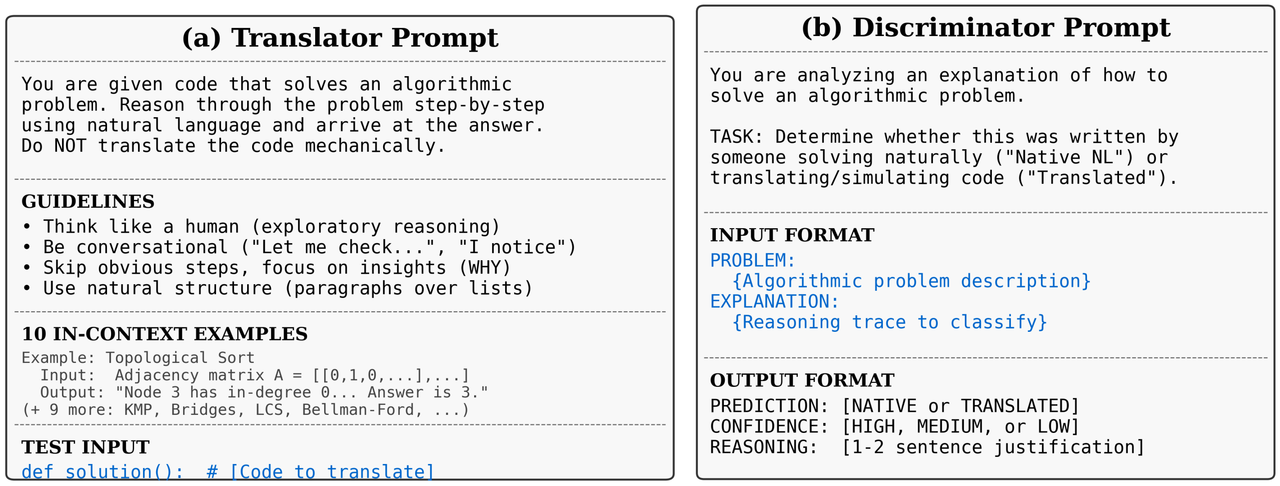
Evaluation Setup We formulate a binary classification task in which a powerful judge model (Claude Opus 4.5) is given a problem instance x, and a single reasoning trace z, and must predict whether z was generated by Arm 1, or the translated Arm 2. This setup corresponds to discriminating between the joint distributions $p(x)p_{NL}(z \mid x)$ and $p(x)p_{Translated}(z \mid x)$. For each evaluation task x, we construct two samples (x, z_{NL}), where $z_{NL} \sim p_{NL}(\cdot \mid X)$, labelled Original, and $(x, \hat{z}_{NL})$ where $\hat{z}_{NL} \sim p_{Translated}(\cdot \mid x)$, labelled Translated. The label counts are balanced on the test set, and the judge's accuracy, and AUC are computed on held-out tasks (disjoint from any in-context prompting examples).

In the evaluation, all generation parameters are fixed, model, prompt, and decoding hyperparameters, and the only variability is the seed for both the original Arm 1 and translated Arm 2, i.e. we compare samples from two fixed conditionals given a task instance.

We ensure the judge has enough discriminative power by asking it to classify between raw code and the native NL reasoning. High accuracy would indicate that the performance results on the main evaluation is not due to an underpowered judge. We prompt the judge zero-shot, instructing it that there are original natural language reasoning traces and translated ones that it must distinguish.

The translator, gemini-2.0 flash, is given 10 in-context pairs between code in Arm 2 and original natural language reasoning in Arm 1, and asked to learn the mapping and apply on new instances. The 10 in-context examples are 10 different tasks, which are disjoint from the 21 other tasks in CLRS benchmark that we evaluate on.

Results. Across 2000 samples, we find that the accuracy of the prompt is $49.4\% \in [47.2\%.51.5\%]$ Wilson CI, obtaining a 0.479 AUC. The control accuracy is 79.0% so the discriminator is calibrated. We include per-task analysis in the appendix. We see that this is generally true across different models with only a few exceptions between rod cutting and the kruskal algorithm problems.



(a) Translator Prompt

```
You are given code that solves an algorithmic
problem. Reason through the problem step-by-step
using natural language and arrive at the answer.
Do NOT translate the code mechanically.
```

GUIDELINES

```
• Think like a human (exploratory reasoning)
• Be conversational ("Let me check...", "I notice")
• Skip obvious steps, focus on insights (WHY)
• Use natural structure (paragraphs over lists)
```

10 IN-CONTEXT EXAMPLES

```
Example: Topological Sort
  Input:  Adjacency matrix A = [[0,1,0,...],...]
  Output: "Node 3 has in-degree 0... Answer is 3."
(+ 9 more: KMP, Bridges, LCS, Bellman-Ford, ...)
```

TEST INPUT

```
def solution():  # [Code to translate]
```

(b) Discriminator Prompt

```
You are analyzing an explanation of how to
solve an algorithmic problem.

TASK: Determine whether this was written by
someone solving naturally ("Native NL") or
translating/simulating code ("Translated").
```

INPUT FORMAT

```
PROBLEM:
  {Algorithmic problem description}
EXPLANATION:
  {Reasoning trace to classify}
```

OUTPUT FORMAT

```
PREDICTION: [NATIVE or TRANSLATED]
CONFIDENCE: [HIGH, MEDIUM, or LOW]
REASONING:  [1-2 sentence justification]
```

Figure 6. Recovery rate analysis: proportion of cases where LLM simulation (Arm 2) produces the correct answer despite incorrect code (Arm 3 failure). Recovery rates remain below 5% across all task families and models, confirming that solver execution dominates LLM simulation.

3.2. Functional Similarity between Translated NL and original NL

We wish to verify whether the translated natural language reaosning from the experiment before has the same functionality as the original natural language reasoning produced in Arm 1.

Evaluation Setup. Given a task instance x, we prompt the target language model (gemini) three ways. 1. Baseline: x (question only), 2. Arm 1: $x||z_{NL}$ 3. Translated Arm 2: $x||\hat{z}_{NL}$ where $\hat{z}_{NL}$ is obtained by translating code to natural language. If the translated NL loses information relative to the original NL, then conditioning on $\hat{z}_{NL}$ should yield worse performance than conditioning on z_{NL}. We run on 1000 samples and report results on held-out tasks disjoint from the ICL example tasks. The same prompt is used for the translator in the previous experiment as the one used here.

Results. We fail to reject the null hypothesis on 1000 samples, and see that using the original Arm 1 and the translated Arm 2 concatenated to the same original task prompt yield the same results.

Qualitative Analysis In observing the LLM responses, we notice the model's have a propensity to follow similar reasoning. This happens across individual models, but problem solving methodology is surprisingly similar across models. Show figure here.

4. Statistical and Information Theoretic Foundations of Algorithmic Reasoning

We claim that natural language reasoning is a garbling of code (**?**) under a noisy channel paradigm of inference, leading code reasoning to be at least as good as NL reasoning. [**Yu**: define/introduce garbling]

We show that the distribution of traces produced directly from the reasoning model can be approximated by post-processing the code with a fixed function across tasks and models. For each task instance x drawn from a held-out test distribution over 21 different algorithmic problems in CLRS, we consider two conditional distributions: Arm 1: $p_{NL}(\cdot \mid x)$, original NL traces generated by an LLM (Gemini 2.0 Flash in our case) and Arm 2 Translated: $p_{Translated}(\cdot \mid x)$, NL traces obtained by first generating code by an LLM, then translating that code into natural language using a fixed translator.[**Yu**: Need fix: this should be simulation not translation. p_{code} is not defined]

Setup. Let $X \sim p(x)$ denote the task instance (problem + inputs), drawn from a representative test distribution. Let $\mathcal{Y}$ be an output space (e.g., answer strings), and let $\ell : \mathcal{Y} \times X \to [0, 1]$ be a bounded and measurable loss function (0–1 binary loss). In Arm 1 and Arm 2, each arm corresponds to an intermediate representation Z (e.g. CoT) produced by channel $p(z \mid x)$ (e.g. LLM), and then choosing an output Y via a randomized decision rule $\delta(y \mid x, z)$ (i.e. LLM test-time reasoning).

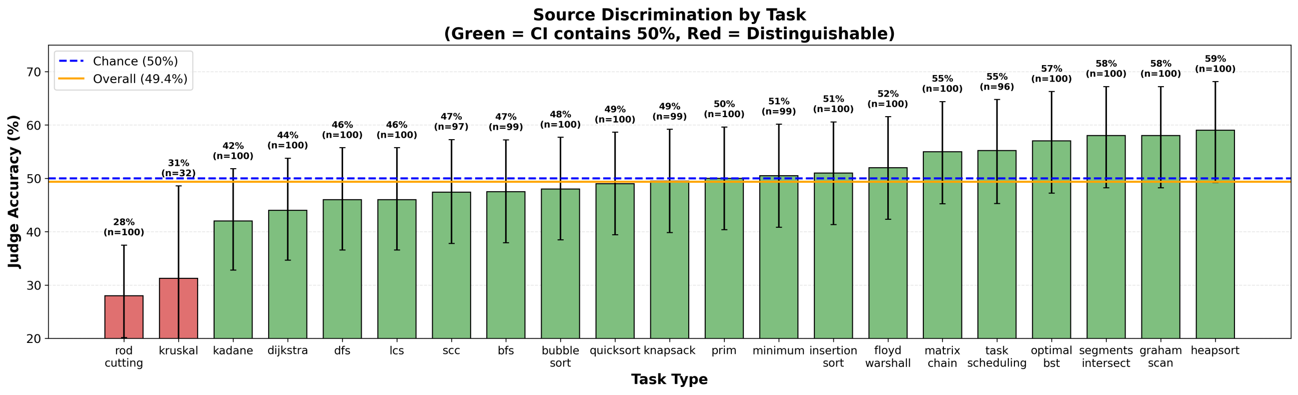


Figure 7. Discrimination accuracy by task: analysis of how well different representations (code vs. natural language) discriminate between algorithm types across task families.

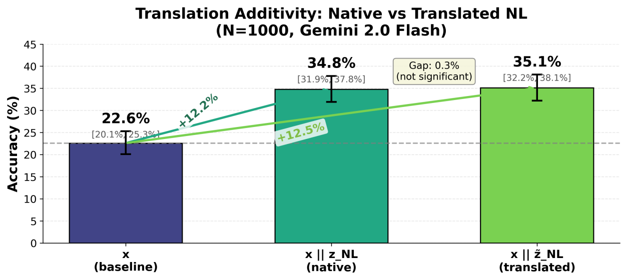


Figure 8. Translation additivity analysis: functional similarity between translated NL and original NL reasoning traces.

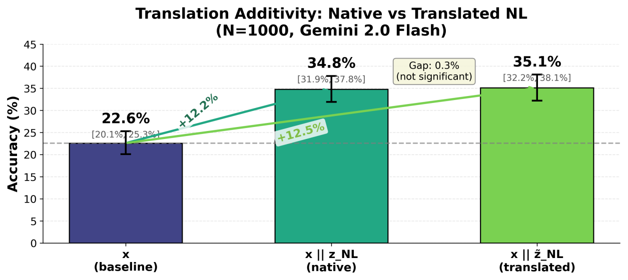


Figure 9. Recovery rate analysis: proportion of cases where LLM simulation (Arm 2) produces the correct answer despite incorrect code (Arm 3 failure). Recovery rates remain below 5% across all task families and models, confirming that solver execution dominates LLM simulation.

For any CoT observation Z, define the Bayes risk:

$$R^*(Z) := \inf_{\delta} \mathbb{E}[\ell(Y, X)],$$
$$X \sim p,\ Z \sim p(\cdot \mid X),\ Y \sim \delta(\cdot \mid X, Z).$$

In the first arm, we observe $Z_{\text{NL}} \sim p_{\text{NL}}(\cdot \mid X)$. For the second arm, we have $Z_{\text{Code}} \sim p_{\text{Code}}(\cdot \mid X)$.

We show that $R^*(Z_{\text{Code}}) \leq R^*(Z_{\text{NL}}) + O(\varepsilon)$ for some negligible ε. [**Yu**: define ε]

4.1. Assumptions

Assumption 1. We assume there exists a stochastic kernel T [**Yu**: define T, citation] such that the Markov chain $X \to Z_{\text{Code}} \to \hat{Z}_{\text{NL}}$ is representative of CoT, with the final decision stage being $\delta(y \mid X, \hat{Z}_{\text{NL}})$. [**Yu**: explain why this can be a markov chain.] That is,

$$\hat{Z}_{\text{NL}} \sim p_{\text{translated}}(\cdot \mid x),\ p_{\text{translated}}(z \mid X)$$
$$:= \int T(z \mid z_{\text{Code}})\, p_{\text{Code}}(z_{\text{Code}} \mid x)\, \mathrm{d}z_{\text{Code}}.$$

Assumption 2. We assume that the original NL reasoning chain of thought is close to the translated NL on average. Let p_{NL} be the Arm 1 channel and $p_{\text{translated}}(\cdot \mid x)$ be the translated NL channel. Assume an average conditional TV bound:

$$\mathbb{E}_{X \sim p}\big[d_{\text{TV}}\big(p_{\text{NL}}(\cdot \mid X),\, p_{\text{translated}}(\cdot \mid X)\big)\big] \leq \varepsilon,$$

where

$$d_{\text{TV}}(P, Q) = \sup_{B} |P(B) - Q(B)|.$$

In other words, averaged over task instances, the NL trace produced by Arm 1 is close in distribution to the NL traces obtained by translating the code trace (Arm 2) using the translator T (Markov kernel).

4.2. Evaluating Translated NL and NL Distributional Similarity.

In this section, we empirically validate the two assumptions stated above. One key hypothesis is that natural language reasoning follows a deeper algorithmic procedure encoded in its representations. If this is the case, it would make sense to surface the code and send it to an executor. We test whether natural language reasoning generated by information contained in the code alone can be distributionally similar to natural language reasoning generated from the prompt alone in Arm 1. Then, we test whether they are functionally similar.

4.2.1. Distributional Similarity between Translated NL and original NL

Evaluation Setup We formulate a binary classification task in which a powerful judge model (Claude Opus 4.5) is given a problem instance x, and a single reasoning trace z, and must predict whether z was generated by Arm 1, or the translated Arm 2. This setup corresponds to discriminating between the joint distributions $p(x)p_{NL}(z \mid x)$ and $p(x)p_{Translated}(z \mid x)$. For each evaluation task x, we construct two samples (x, z_{NL}), where $z_{NL} \sim p_{NL}(\cdot \mid X)$, labelled Original, and $(x, \hat{z}_{NL})$ where $\hat{z}_{NL} \sim p_{Translated}(\cdot \mid x)$, labelled Translated. The label counts are balanced on the test set, and the judge's accuracy, and AUC are computed on held-out tasks (disjoint from any in-context prompting examples).

In the evaluation, all generation parameters are fixed, model, prompt, and decoding hyperparameters, and the only variability is the seed for both the original Arm 1 and translated Arm 2, i.e. we compare samples from two fixed conditionals given a task instance.

We ensure the judge has enough discriminative power by asking it to classify between raw code and the native NL reasoning. High accuracy would indicate that the performance results on the main evaluation is not due to an underpowered judge. We prompt the judge zero-shot, instructing it that there are original natural language reasoning traces and translated ones that it must distinguish.

The translator, gemini-2.0 flash, is given 10 in-context pairs between code in Arm 2 and original natural language reasoning in Arm 1, and asked to learn the mapping and apply on new instances. The 10 in-context examples are 10 different tasks, which are disjoint from the 21 other tasks in CLRS benchmark that we evaluate on.

Results. Across 2000 samples, we find that the accuracy of the prompt is 49.4% $\in$ [47.2%.51.5%] Wilson CI, obtaining a 0.479 AUC. The control accuracy is 79.0% so the discriminator is calibrated. We include per-task analysis in the appendix. We see that this is generally true across different models with only a few exceptions between rod cutting and the kruskal algorithm problems.

4.2.2. Functional Similarity between Translated NL and original NL

We wish to verify whether the translated natural language reaosning from the experiment before has the same functionality as the original natural language reasoning produced in Arm 1.

Evaluation Setup. Given a task instance x, we prompt the target language model (gemini) three ways. 1. Baseline: x (question only), 2. Arm 1: $x||z_{NL}$ 3. Translated Arm 2: $x||\hat{z}_{NL}$ where $\hat{z}_{NL}$ is obtained by translating code to natural language. If the translated NL loses information relative to the original NL, then conditioning on $\hat{z}_{NL}$ should yield worse performance than conditioning on z_{NL}. We run on 1000 samples and report results on held-out tasks disjoint from the ICL example tasks. The same prompt is used for the translator in the previous experiment as the one used here.

Results. We fail to reject the null hypothesis on 1000 samples, and see that using the original Arm 1 and the translated Arm 2 concatenated to the same original task prompt yield the same results.

Qualitative Analysis In observing the LLM responses, we notice the model's have a propensity to follow similar reasoning. This happens across individual models, but problem solving methodology is surprisingly similar across models. Show figure here.

4.3. Proof

Proof. Under Assumptions 1–2, for the bounded loss $\ell \in [0, 1]$,

$$R^*(Z_{\mathrm{Code}}) \leq R^*(Z_{\mathrm{NL}}) + \varepsilon.$$

Step 1: Simulate NL from code via translation. Here we first translate the input problem into CoT, then execute the CoT:

$$\delta_{\mathrm{Code}}(y \mid x, z_{\mathrm{Code}}) := \int \underbrace{\delta_{\mathrm{NL}}(y \mid x, z)}_{\text{Execute}} \underbrace{T(z \mid z_{\mathrm{Code}})}_{\text{Translate}} \, \mathrm{d}z.$$

Let $Y_{\mathrm{Code}} \sim \delta_{\mathrm{Code}}(\cdot \mid X, Z_{\mathrm{Code}})$. Let $\hat{Y}_{\mathrm{translated}} \sim \delta_{\mathrm{NL}}(\cdot \mid X, \hat{Z}_{\mathrm{NL}})$, where $\hat{Z}_{\mathrm{NL}}$ is produced from Z_{Code} via the translator kernel T.

The joint distributions (X, Y_{Code}) and $(X, \hat{Y}_{\mathrm{translated}})$ are the same. Thus,

$$\mathbb{E}[\ell(Y_{\mathrm{Code}}, X)] = \mathbb{E}[\ell(\hat{Y}_{\mathrm{translated}}, X)].$$

This is because conditional on $X = x$, sampling $Z_{code} \sim p_{code}(\cdot \mid x)$, then $\hat{Z}_{nl} \sim T(\cdot \mid Z_{code})$, then $Y \sim \delta_{NL}(\cdot \mid x, \hat{Z}_{NL})$ induces the same conditional distribution over Y as $Y \sim \delta_{code}(\cdot \mid x, Z_{code})$.

Step 2: Substitute translated NL and original NL via TV lemma.

Lemma 4.0.1 (TV Lemma). *Let $X \sim p(x)$. Let $Z \mid X = x \sim P_x$ and $Z' \mid X = x \sim Q_x$. Let $g(x, z) \in [0, 1]$ be* *measurable. Then*

$$\mathbb{E}[g(X,Z)] - \mathbb{E}[g(X,Z')] \le \mathbb{E}_X\left[d_{\mathrm{TV}}(P_X, Q_X)\right].$$

Suppose we have $Y_{\mathrm{NL}} \sim \delta_{\mathrm{NL}}(\cdot \mid Z_{\mathrm{NL}})$ under the actual channel $p_{\mathrm{NL}}(\cdot \mid x)$. For each x and trace z, define $g(x,z) := \mathbb{E}_{y\sim\delta(\cdot|z)}[\ell(y,x)]$.

Then $g(x,z) \in [0,1]$. Note that

$$\mathbb{E}[\ell(Y_{\mathrm{NL}},X)] = \mathbb{E}[g(X,Z_{\mathrm{NL}})],$$
$$\mathbb{E}[\ell(\hat{Y}_{\mathrm{translated}},X)] = \mathbb{E}[g(X,\hat{Z}_{\mathrm{NL}})].$$

Applying the TV lemma with $P_x = p_{\mathrm{NL}}(\cdot \mid x)$ and $Q_x = p_{\mathrm{translated}}(\cdot \mid x)$:

$$\begin{aligned}&\left|\mathbb{E}[\ell(Y_{\mathrm{NL}},X)] - \mathbb{E}[\ell(\hat{Y}_{\mathrm{translated}},X)]\right| \\ &\quad= \left|\mathbb{E}[g(X,Z_{\mathrm{NL}})] - \mathbb{E}[g(X,\hat{Z}_{\mathrm{NL}})]\right| \\ &\quad\le \mathbb{E}_X\left[d_{\mathrm{TV}}(p_{\mathrm{NL}}(\cdot \mid X),\, p_{\mathrm{translated}}(\cdot \mid X))\right] \le \varepsilon.\end{aligned}$$

Therefore, rearranging gives

$$\mathbb{E}[\ell(\hat{Y}_{\mathrm{translated}},X)] \le \mathbb{E}[\ell(Y_{\mathrm{NL}},X)] + \varepsilon.$$

Thus,

$$\mathbb{E}[\ell(Y_{\mathrm{Code}},X)] = \mathbb{E}[\ell(\hat{Y}_{\mathrm{translated}},X)] \le \mathbb{E}[\ell(Y_{\mathrm{NL}},X)] + \varepsilon.$$

Since this holds for arbitrary NL rule δ_{NL}, taking the infimum over δ_{NL} on the right-hand side yields $R^*(Z_{\mathrm{Code}}) \le R^*(Z_{\mathrm{NL}}) + \varepsilon$. □

4.4. Arm 2 < Arm 3

We model the comparison between LLM simulation (Arm 2) and solver execution (Arm 3) under the following assumptions:

1. **Solver correctness.** Given correct code Z, the solver deterministically produces the ground-truth answer: $Y_3 = g(X,Z) = Y^*(X)$.
2. **Noisy simulation.** LLM simulation adds stochastic noise: $Y_2 \sim N(\cdot \mid Y_3, X, Z)$ for some noise kernel N.
3. **0-1 loss.** We evaluate under $\ell(y,x) = \mathbf{1}\{y \neq Y^*(x)\}$.

Under these assumptions, the solver achieves zero risk ($R_3 = 0$) whenever the generated code is correct. In contrast, LLM simulation incurs positive risk ($R_2 > 0$) whenever $\Pr[Y_2 \neq Y_3] > 0$, which occurs empirically due to execution errors in mental simulation. Therefore, $R_3 < R_2$.

The only scenario where Arm 2 could outperform Arm 3 is when the generated code is *incorrect*, yet the LLM “recovers” by reasoning to the correct answer despite the flawed code. We empirically quantify this recovery rate below.

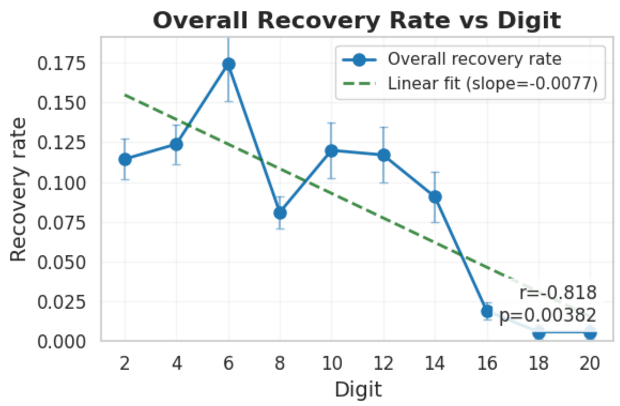


Figure 10. Recovery rate analysis: proportion of cases where LLM simulation (Arm 2) produces the correct answer despite incorrect code (Arm 3 failure). Recovery rates remain below 5% across all task families and models, confirming that solver execution dominates LLM simulation.

Recovery reduces as tasks get harder. To further reinforce this result, we rule out the possibilities of recovery as tasks get harder, eliminating any benefit of running Arm 2:

1. Arm 3 produces an incorrect answer (implying incorrect code generation), and
2. Arm 2 produces the correct answer (implying successful LLM recovery).

Figure 10 presents the recovery analysis across all tasks and models. The recovery rate remains consistently low (typically $< 5\%$), indicating that LLM simulation rarely compensates for code generation errors. This confirms that Arm 3’s advantage stems from reliable solver execution rather than Arm 2’s inability to reason about code.

5. Related Work and Discussion

Neuro-symbolic Learning. This paper builds on research in neuro-symbolic integration (**????**), which combines neural networks with symbolic reasoning systems. These approaches are motivated by cognitive science (**???**), hierarchical reinforcement learning (**??**), and compositionality research (**????**). An orthogonal line of work explores direct execution of algorithms by neural networks (**????**). Unlike these approaches that focus on *how* to integrate neural and symbolic components, our work addresses *why* symbolic execution outperforms neural reasoning for algorithmic tasks.

LLM Reasoning. Recent work has explored various reasoning paradigms for LLMs, including symbolic reasoning

(**???**), chain-of-thought prompting (**????**), and in-context learning (**????**). **?** model in-context learning as implicit Bayesian inference, which we extend to compare different reasoning representations. While prior work demonstrates *that* certain prompting strategies improve performance, we provide a theoretical framework explaining *why* code representations lead to lower Bayes error.

LLM Tool-Use. Tool-augmented LLMs have achieved strong empirical results (**?????**). Code generation for tool-use can be viewed as a form of semantic parsing (**????**) or function calling (**???**). Our work complements this literature by providing theoretical justification for the observed empirical advantages of code-based tool-use over direct natural language reasoning.

6. Conclusion

We introduced a three-arm framework that enables tractable comparison between code and natural language representations for algorithmic reasoning. By modeling LLM inference as Bayesian inference, we proved that code representations yield higher mutual information with target algorithms, leading to lower Bayes error. Empirically, code execution achieves 78% accuracy compared to 21% for natural language reasoning across arithmetic, dynamic programming, and integer linear programming tasks.

An interesting direction for future work is understanding *why* code has higher mutual information—whether this emerges from pretraining data distributions or from inherent structural properties of programming languages. Our framework provides a foundation for such investigations.

Limitations.

Our study focuses on algorithmic tasks (arithmetic, DP, ILP) where ground truth is well-defined. The results may not generalize to open-ended reasoning tasks without clear algorithmic structure. Additionally, our theoretical bounds are asymptotic—the 6% Bayes error improvement is a lower bound that may not reflect finite-sample performance. Finally, we evaluate on a limited set of models (Deepseek, Gemma); behavior may differ for other architectures.

Future Work. These findings have practical implications for AI system design: for algorithmically structured problems, compositional systems with symbolic execution should be preferred over monolithic neural reasoning. This supports the growing trend toward tool-augmented LLMs.