Code as Explanations: Executable and Verifiable LLM Explanations

Nhi Nguyen
nhi.nguyen@nyu.edu

Shauli Ravfogel
shauli.ravfogel@nyu.edu

Rajesh Ranganath
rajeshr@cims.nyu.edu

Abstract

As large language models (LLMs) increasingly make consequential decisions and solve problems alongside humans, we must understand how models arrive at their conclusions. Understanding how an LLM behaves requires observing it across multiple inputs, yet free-text explanations such as chain-of-thought (CoT) are generated per instance and cannot be easily verified on new inputs. We propose *code explanations* to address the mismatch in how LLM explanations are generated versus how they are evaluated: a program explains LLM behavior across a dataset of inputs, and its execution enables automatic verification. We define CONSISTENCY metric that measures the fraction of inputs on which the program reproduces the LLM's outputs, and use it to evaluate and select candidate programs. Across three open-source code-capable LLMs and eight tasks, we find that LLMs can produce highly consistent code explanations in several settings. Code explanations are also $6\times$ to $100\times$ shorter than CoT, and better align with model behavior in 9 out of 24 settings. However, consistency is significantly lower on tasks with more complex linguistic components, where generated programs reproduce model outputs no more frequently than an uninformative explanation baseline, and these programs often rely on longer hard-coded patterns. In experiments with Claude Opus 5.5, providing the model with consistency feedback and code execution improves explanation consistency across tasks, but consistency remains significantly lower on more complex tasks. Our findings establish executable code as a novel, verifiable, and compact approach to explaining LLM behavior and highlight verification feedback as a promising direction for improving explanation consistency. The code for this paper is available at `https://github.com/rajesh-lab/code-as-explanations`.

1 Introduction

Large language models (LLMs) are increasingly solving problems alongside humans in high-stakes settings, from predicting hospital readmission from clinical notes [17], aiding legal decision-making from case documents [37], to predicting properties of molecular sequences [19]. As LLMs take on more consequential tasks, understanding how models arrive at their decisions is critical both for detecting undesirable behaviors such as reward hacking [34, 44, 47] and for enabling humans to learn novel problem-solving strategies from models' reasoning [18, 31].

A prominent approach to understanding LLMs is to ask models to explain their reasoning in free text, commonly via chain-of-thought (CoT) [26, 49] or post-hoc rationales [20, 21]; for example, OpenAI audited the reasoning traces of agents that reward-hacked the ExploitGym benchmark to reveal unexpected coordination behavior [34]. Such explanations are only useful if they accurately capture model behavior. However, evaluating free-text explanations is challenging, as these explanations are tied to individual instances, while understanding LLM behavior requires observing the model's outputs across multiple inputs. Existing LLM explanation evaluations therefore often rely on targeted input perturbations [5, 28, 45], and Nguyen et al. [33] show that explanation sufficiency can change depending on the chosen input distribution.

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

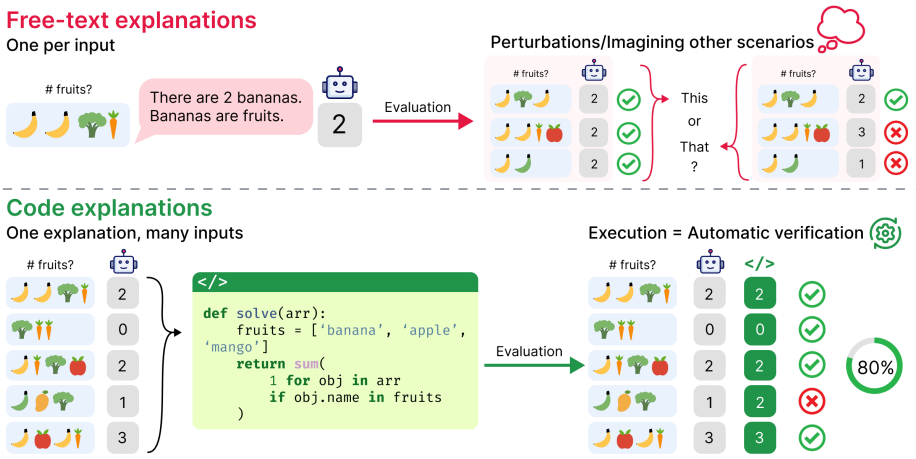


Figure 1: Overview of code explanations. Free-text explanations describe LLM behavior on individual inputs, and require input perturbations to evaluate whether they capture model behavior. In contrast, code explanations represent model behavior across multiple inputs as executable programs, enabling automatic verification by comparing program outputs with LLM outputs.

To bridge the gap between how LLM explanations are generated and how they are evaluated, we propose explaining LLMs with executable code (Figure 1 and Section 3). A program can be executed on arbitrary inputs, allowing automatic verification by comparing its outputs with those of the LLM. Current LLMs are also well equipped to generate code, having been extensively pretrained on code data [51]. In addition, when compared with a collection of instance-level CoT, a single compact code explanation reduces the cognitive load required for humans to understand model behavior. To evaluate code explanations, we introduce CONSISTENCY, which quantifies how often an explanation reproduces an LLM's outputs over a dataset (Section 4.1). CONSISTENCY can be computed efficiently by executing the explanation on each input once, enabling a generation procedure where we sample multiple candidate programs and select the most consistent code explanation (Section 4.2).

We use this procedure to study the empirical code explanation consistency across three open-source, code-capable LLMs and eight tasks spanning different levels of linguistic complexity (Section 5). Experimental results show that LLMs can produce highly consistent code explanations in several model-task settings. Compared to collections of CoT, code explanations are 6× to 100× shorter when explaining the same dataset, while better aligning with LLM behavior in 9 out of 24 settings, spanning both simpler and more complex tasks (Section 5.3). However, high consistency is not guaranteed, even when task accuracy is high. On tasks involving more complex linguistic features, code explanations achieve consistency comparable to an uninformative majority-answer baseline (Section 5.1), and generated programs tend to rely on longer hard-coded linguistic patterns (Section 5.2). We further investigate whether verification feedback can improve explanation consistency by either providing non-agentic LLMs with multi-turn consistency feedback and/or equipping LLM agent Claude Opus 5.5 with code execution and consistency-evaluation tools (Section 5.4). Both multi-turn refinement and code execution improve consistency, particularly on simpler logical tasks, but even Claude struggles to produce consistent code explanations on more complex tasks.

Our findings demonstrate the advantages of code as a verifiable and compact approach to explaining LLM behavior, while showing that reliably generating consistent explanations remains an open problem, particularly for complex linguistic tasks. Code execution and our CONSISTENCY metric offer promising directions for future optimization of code explanations through verification feedback.

2 Related work

Free-text explanations. Free-text explanations such as chain-of-thought (CoT) [49] are widely used to both improve task performance [49, 54] and provide insight into model behaviors [6, 25]. To evaluate whether free-text explanations accurately reflect the model's internal reasoning, existing evaluations often rely on input perturbations, e.g., by inserting hints or biases [2, 8, 45] or generating

alternative inputs with an LLM [28, 33]. Parcalabescu and Frank [35] argue that we can only evaluate LLM explanations via consistency across multiple model outputs, and Nguyen et al. [33] prove that explanation sufficiency can change depending on the chosen input distribution. We show that assessing the alignment between an explanation and LLM behavior requires observing model outputs across multiple inputs, and we propose code as a new LLM explanation paradigm where code execution enables direct verification against model outputs on arbitrarily many inputs.

LLMs and code generation. Code data is prominent in LLM pre-training corpora [51], and recent advances in code-capable LLMs have led to strong performance across programming benchmarks, including function-level tasks [3, 7], class-level or multi-file code [10, 53], and competitive programming problems [12, 14, 50]. Prior work has studied code as a task output [16] and as an action space for LLM agents [48]. We instead propose using code as a novel and verifiable form of LLM explanation.

Sparse, compact explanations of dense models. Machine learning models can be broadly categorized into two paradigms: *sparse models* (generalized linear models [32], decision trees [9]), which use a small number of simple operations that humans can understand; and *dense models* (CNN [23], transformers [46]), whose large number of parameters and computations are opaque to humans. Explaining dense models like LLMs thus involves finding sparse, compact descriptions or approximations of their computations, such as sparse circuits [27], behavior-associated neurons [41], and chain-of-thought traces [49]. For traditional explanation methods such as feature attributions, sparsity - the number of selected features - is also a desideratum for interpretability [15, 52]. In our work, we use several code complexity metrics to measure the sparsity and compactness of code explanations as proxies for their interpretability to humans. We also compare the compactness of code and free-text explanations using their token counts.

3 Code explanations

3.1 Free-text explanations

We first review instance-wise free-text explanations. Let $\mathbf{x} \in \mathcal{X}$ denote the input, $\mathbf{y} \in \mathcal{Y}$ denote the output, and let $F(\mathbf{y} \mid \mathbf{x})$ denote the output distribution of a model F on input $\mathbf{x}$ given a task. Let $g(\mathbf{e} \mid \mathbf{x})$ denote an explanation method that maps input $\mathbf{x}$ to a distribution of explanations $\mathbf{e} \in \mathcal{E}$. Given an input instance $\boldsymbol{x}$, an explanation $\boldsymbol{e} \sim g(\mathbf{e} \mid \boldsymbol{x})$ describes which properties of $\boldsymbol{x}$ influence model F's output and how the model uses them to produce that output.

One reason free-text explanations are widely used is their apparent simplicity: one explanation describes one input-output pair. However, evaluating whether a free-text explanation accurately captures the model's output-generating process requires more than the original input-output pair. Existing evaluations specify certain input perturbations to assess how the model behaves under changes to the input. For example, [2, 8, 33, 45] alter input properties that the explanation deems unimportant and test whether the model's output remains the same. Conversely, [28, 29] modify properties deemed important by the explanation and identify the minimal edits needed to change the model's output.

The need for additional inputs to evaluate instance-wise explanations is unavoidable. Observing the original input-output pair alone cannot establish whether an input property influences the model's output, as formalized by the following proposition.

Proposition 3.1 (Unidentifiability from observations)**.** *Let $\gamma : \mathcal{X} \to \mathcal{G}$ be an input property function and let $f : \mathcal{X} \to \mathcal{Y}$ be an output function. Suppose that f can only be observed on a proper subset $\mathcal{S} \subset \mathcal{X}$, where $\gamma(\mathbf{x}) = C$ for all $\mathbf{x} \in \mathcal{S}$. Consider two candidate output functions:*

1. *$f_1 : \mathcal{X} \to \mathcal{Y}$ that does not depend on γ;*
2. *$f_2(\mathbf{x}) = \begin{cases} f_1(\mathbf{x}) & \text{if } \gamma(\mathbf{x}) = C \\ f'(\mathbf{x}) & \text{if } \gamma(\mathbf{x}) \neq C \end{cases}$, where $f'(\mathbf{x}) \neq f_1(\mathbf{x})$ whenever $\gamma(\mathbf{x}) \neq C$.*

Although f_1 does not depend on γ and f_2 does, they agree on every input in $\mathcal{S}$. Thus, whether the output function f depends on the property γ cannot be identified from observations in $\mathcal{S}$.

In other words, if an input property is constant across all observed inputs, its influence on the model's outputs cannot be determined from those observations. In particular, when we only observe a single

input (i.e., $|\mathcal{S}| = 1$), as in the case of instance-wise free-text explanations, we cannot determine the influence of any input property on the model's outputs.

This result highlights a fundamental complexity of free-text explanations: although they provide a seemingly simple, one-to-one description of each input-output pair, **evaluating whether free-text explanations capture the model behavior necessarily requires additional inputs.** This introduces two challenges. First, free-text explanations are instance-specific, so evaluating them requires specifying an input distribution $q_{\mathbf{x}}$ that the explanation method itself does not provide. Second, a free-text explanation does not specify model behavior on arbitrary inputs, so its evaluation is restricted to either inputs that produce the same explanation or minimally perturbed inputs that change the model's output. These challenges are reflected in existing metrics such as *sufficiency* [33], which explicitly requires an input distribution and evaluates model behavior over alternative inputs that yield the same explanation. Given explanation method $g(\mathbf{e} \mid \mathbf{x})$ and an input distribution $q_{\mathbf{x}}$, then $q_{\mathbf{x} \mid \mathbf{e}}$ denotes the distribution over alternative inputs that produce the same explanation. Sufficiency compares the original output distribution $F(\mathbf{y} \mid \mathbf{x})$ with the explanation-dependent distribution $A_{\mathbf{e}}(\mathbf{y} \mid \mathbf{x}) = \int F(\mathbf{y} \mid \mathbf{x}') q_{\mathbf{x} \mid \mathbf{e}}(\mathbf{x}' \mid \mathbf{e})\, d\mathbf{x}'$. Evaluating sufficiency thus requires specifying $q_{\mathbf{x}}$ and $q_{\mathbf{x} \mid \mathbf{e}}$.

To address these challenges, we introduce code explanations that are executable and thus can be evaluated against LLM behavior on arbitrary inputs from a task distribution.

3.2 Code as executable and verifiable explanations

We define an *executable code explanation* as a single program that describes an LLM's behavior on a task. Formally, let $\mathtt{T}$ denote a task with input distribution $q_{\mathbf{x}}$. Let $\mathcal{D} = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^{n} \sim (q_{\mathbf{x}} \cdot F(\mathbf{y} \mid \mathbf{x}))^{\otimes n}$ be a dataset of input-output observations sampled from the model. We define a code explanation method $h(c \mid \mathcal{D})$ to be a function that maps observations $\mathcal{D}$ to a distribution over executable programs c. A program may be deterministic or stochastic. We denote its output distribution on input $\mathbf{x}$ as $c(\mathbf{y} \mid \mathbf{x})$.

Consequently, a code explanation directly specifies an explanation-dependent distribution $A_c(\mathbf{y} \mid \mathbf{x}) := c(\mathbf{y} \mid \mathbf{x})$ that can be compared against the original LLM without any additional specification. A code explanation $c \sim h(c \mid \mathcal{D})$ is *consistent* with model F on task $\mathtt{T}$ if:

$$F(\mathbf{y} \mid \mathbf{x}) = c(\mathbf{y} \mid \mathbf{x}) \qquad q_{\mathbf{x}}\text{-almost surely.} \tag{1}$$

In words, consistency requires the code explanation to reproduce the model's output distributions across inputs from the task distribution.

Both free-text and code explanations can be evaluated by comparing their explanation-dependent distributions, $A_{\mathbf{e}}(\mathbf{y} \mid \mathbf{x})$ and $A_c(\mathbf{y} \mid \mathbf{x})$ respectively, with the original LLM behavior. However, code explanations directly specify executable functions, allowing their behavior to be evaluated on arbitrary inputs from the task distribution without specifying a separate explanation-conditioned distribution $q_{\mathbf{x} \mid \mathbf{e}}$. This addresses the mismatch between how LLM explanations are generated and evaluated. When explaining a dataset, a single program can also provide a more compact and thus human-understandable description of model behavior than a large collection of instance-wise free-text explanations. These properties make code explanations an automatically verifiable and compact alternative to instance-wise free-text explanations.

4 Evaluating and generating code explanations

4.1 Evaluation

We build on the formalization of executable explanations in Section 3.2 to derive a metric for evaluating the consistency of code explanations for LLMs. We focus on the case where the program c is deterministic, as is typical for code generated by LLMs. In this setting, we define a metric that measures how well the program c aligns with the model's output-generating process $F(\mathbf{y} \mid \mathbf{x})$ by measuring how often, over a dataset, the program produces the same output as the model. Formally, let $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{D} \sim (q_{\mathbf{x}} \cdot F(\mathbf{y} \mid \mathbf{x}))^{\otimes D}$ be a dataset of input-model output pairs, and let $c(\cdot)$ be an executable program that maps an input $\boldsymbol{x}$ to an output $\boldsymbol{y}$. We define code consistency as:

$$\textsc{Consistency}(c, \mathcal{D}) := \frac{1}{D} \sum_{i=1}^{D} \mathbf{1}\big[c(\boldsymbol{x}_i) = \boldsymbol{y}_i\big]. \tag{2}$$

Algorithm 1: Generate code explanation

Input: Large language model F; input dataset $\{\boldsymbol{x}_i\}_{i=1}^{D}$; task instruction $\texttt{T}$; explanation prompt $\texttt{C}$; # in-context examples N; # explanations per example set M; # example sets P
Output: Code explanation c; code consistency
Construct dataset $\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i \sim F(\mathbf{y} \mid \boldsymbol{x}_i; \texttt{T})\}_{i=1}^{D}$.
Partition $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{test}}$ and $\mathcal{D}_{\text{train}} \cap \mathcal{D}_{\text{test}} = \varnothing$.
Sample P subsets $\{\mathcal{D}^{(i)} \subset \mathcal{D}_{\text{train}}\}_{i=1}^{P}$, each subset with size $|\mathcal{D}^{(i)}| = N$.
For each $\mathcal{D}^{(i)}$, sample M candidates $\{c^{(i,j)}\}_{j=1}^{M} \sim F(\mathbf{c} \mid \mathcal{D}^{(i)}; \texttt{T}, \texttt{C})^{\otimes M}$.
Select
$\mathcal{C}^\star = \left\{ c^{(i,j)} : \textsc{Consistency}\left(c^{(i,j)}, \mathcal{D}_{\text{train}}\right) = \max_{i',j'} \textsc{Consistency}\left(c^{(i',j')}, \mathcal{D}_{\text{train}}\right) \right\}$.
Compute $\texttt{consistency} = \frac{1}{|\mathcal{C}^\star|} \sum_{c \in \mathcal{C}^\star} \textsc{Consistency}(c, \mathcal{D}_{\text{test}})$.
Return: $\mathcal{C}^\star$; $\texttt{consistency}$

This metric applies to any deterministic code explanation c. The explanation may be generated by the same LLM that produces the output $\mathbf{y}$, by a different LLM, or written by a human. In this paper, we focus on evaluating self-explanations produced by the same LLM.

Since datasets can contain different numbers of possible outputs, we also define a metric that is normalized against an uninformative explanation that always guesses the most common output. This allows us to compare code explanation consistency across different model-task settings with different output spaces and distributions.

$$\Delta\textsc{Consistency}(c, \mathcal{D}) = \frac{1}{D}\sum_{i=1}^{D} \mathbf{1}\left[c(\boldsymbol{x}_i) = \boldsymbol{y}_i\right] - \frac{1}{D}\left(\max_{1 \le k \le D} \sum_{i=1}^{D} \mathbf{1}\left[\boldsymbol{y}_i = \boldsymbol{y}_k\right]\right). \tag{3}$$

Both $\textsc{Consistency}$ and $\Delta\textsc{Consistency}$ can be evaluated efficiently, requiring only a single execution of c on each input $\boldsymbol{x}$ and comparison with the original model's output.

4.2 Generation

Building on the automatic verification of code explanations in Section 4.1, we introduce a generation procedure that produces multiple candidate code explanations and selects the most consistent one. Let $F(\mathbf{y} \mid \mathbf{x})$ be an LLM and let $\{\boldsymbol{x}_i\}_{i=1}^{D} \sim q_{\mathbf{x}}$ be a dataset of inputs with a corresponding task instruction $\texttt{T}$. We first construct a dataset of input-model output pairs:

$$\mathcal{D} = \{\boldsymbol{x}_i, \boldsymbol{y}_i \sim F(\mathbf{y} \mid \boldsymbol{x}_i; \texttt{T})\}_{i=1}^{D}, \tag{4}$$

where $F(\mathbf{y} \mid \boldsymbol{x}_i; \texttt{T})$ denotes the LLM's output distribution give input $\boldsymbol{x}_i$ and task instruction $\texttt{T}$. We then partition $\mathcal{D}$ into disjoint training and test sets, $\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{test}}$ and $\mathcal{D}_{\text{train}} \cap \mathcal{D}_{\text{test}} = \varnothing$. We hold out $\mathcal{D}_{\text{test}}$ from both generation and selection of the code explanation to avoid optimistically biased estimates of the selected program's consistency on new inputs.

On $\mathcal{D}_{\text{train}}$, we repeatedly sample a subset of N input-model output pairs as in-context examples, then prompt the LLM to generate a program c that explains how the model produces the corresponding outputs. See Table 3 for an example prompt. Using subsets of the training set as in-context examples avoids exceeding the LLM's context-length limit and encourages the LLM to produce programs that generalize beyond the examples. Each subset provides a held-out set for evaluating and selecting generated programs. We use stochastic decoding to generate multiple candidate programs for each sampled subset. All candidates are evaluated on the full $\mathcal{D}_{\text{train}}$, and the programs with the highest $\textsc{Consistency}$ are selected. Finally, we evaluate the selected programs on $\mathcal{D}_{\text{test}}$ and return these programs and their average consistency. The full procedure is summarized in Algorithm 1.

5 Experiments

We use our evaluation and generation methods to study code explanation consistency across a range of language models and language tasks.

Table 1: Overview of the eight BBH [40] tasks used in our evaluation, grouped by reasoning type and linguistic complexity, accompanied by a brief task description.

Category	Task	Description
Structured logic	Web of lies	Determine if a person lies based on a set of statements
	Navigate	Track movements on grid to decide if agent returns to the start
Logic + simple language	Hyperbaton	Identify the sentence with the correct adjective order
	Object counting	Count the number of objects satisfying a given description
Logic + complex language	Formal fallacies	Determine whether an argument is deductively valid or invalid
	Disambiguation QA	Identify the antecedent of a pronoun
Linguistic reasoning	Snarks	Identify which statement is sarcastic
	Translation error	Identify errors in a translation and specify their type

Models. We study code-based explanations using three open-source, instruction-tuned, code-capable language models: Gemma3-27B [42], Qwen2.5-Coder-32B [13], and Llama3.1-70B [11], selected based on their strong performance on standard code generation benchmarks HumanEval [7] and MBPP [3]. They also span different parameter scales and model families, enabling a diverse evaluation. In Section 5.4, we study Claude Opus 5.5 [1] to test a frontier coding agent.

Tasks. To evaluate code explanation consistency, we consider eight tasks from the BIG-Bench Hard (BBH) benchmark [40]. We group these tasks into four categories based on the type of reasoning and linguistic complexity required to solve them: (i) structured logic, (ii) logic with simple language components, (iii) logic with complex language components, and (iv) linguistic reasoning. We describe each task in Table 1. See Section C for a representative input and question for each task. These four categories provide a coarse characterization of task complexity, allowing us to examine how code explanations perform across different levels of linguistic complexity.

Implementation details. We follow Algorithm 1 using the prompt in Table 3. For each dataset, we sample $P = 10$ sets of $N = 20$ in-context examples and generate $M = 10$ explanations per set, yielding 100 explanations in total. We choose these fixed hyperparameters to provide a representative evaluation of code explanations; we leave a more comprehensive study of parameter choices to future work. Hereafter, we use *code consistency* to refer to the test consistency returned by Algorithm 1. Model answers are generated deterministically with chain-of-thought, while code explanations are sampled stochastically. We compare code outputs with model outputs using fuzzy matching. Unless otherwise specified, experimental results are averaged over three random seeds. Further implementation details are provided in Section B.

In the following sections, we consider five research questions about LLM code explanations across our experiments: **(Q1)** Can LLMs explain their behaviors consistently with code explanations? **(Q2)** Which tasks are more difficult for LLMs to generate consistent code explanations? **(Q3)** How do generated code explanations differ across task difficulty? **(Q4)** How do code explanations compare with free-text explanations? **(Q5)** Can code explanation consistency be improved?

5.1 Code consistency across models and tasks

We measure code consistency along with task accuracy across model-task settings in the top panel of Figure 2. Code consistency exceeds 0.75 in 6 of 24 model-task settings, indicating that LLMs can produce highly consistent code explanations in several settings. However, these comprise only 25% of settings, suggesting that high code consistency is possible but not guaranteed, providing a partially positive answer to **(Q1)**. High task accuracy is not correlated with high code consistency, suggesting that consistency reflects a model's ability to explain its behavior rather than simply its ability to solve the task. Figure 7 further stratifies consistency by answer correctness. In over half of the cases, code explanations are no more than $+0.1$ more consistent for correct than incorrect answers. So the selected programs do not systematically explain only the model's correct predictions. They attempt to explain the model's behavior regardless of answer correctness, rather than simply following task instructions.

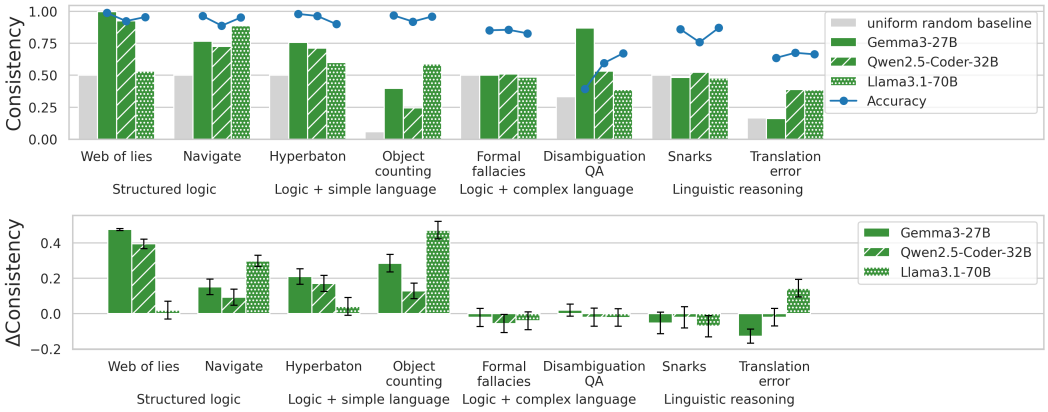


Figure 2: **(Top)** CONSISTENCY and task accuracy, with a uniform random baseline for comparison across tasks with different output-space sizes. Consistency is not correlated with task accuracy, and 5 of the 6 most consistent explanations occur in the first four, simpler tasks. **(Bottom)** The average ΔCONSISTENCY with 95% CI. In the first four tasks, at least one model achieves ΔCONSISTENCY ≥ 0.2, while all but one setting in the last four tasks have CIs that include or fall below zero.

We next examine whether task difficulty affects code explanation consistency (**Q2**). In the top panel of Figure 2, 5 of the 6 highest-consistency cases occur among the first four tasks. The bottom panel, which plots averages ΔCONSISTENCY with 95% confidence intervals (CI), shows a similar pattern: for each of the first four tasks, at least one model achieves average consistency $\geq +0.2$ above the uninformative majority-answer baseline. In contrast, all but one of the last four task-model settings have CI that includes or falls entirely below zero. These results suggest that tasks with more complex language components are more difficult for LLMs to generate consistent code explanations, providing an answer to **(Q2)**. The only high-consistency case among the last four tasks is Gemma on Disambiguation QA. However, its zero ΔCONSISTENCY and low task accuracy suggest that Gemma predicts the majority answer rather than solving the task, making its behavior easier for code explanations to capture.

5.2 Inspect most consistent code explanations

We ask whether more complex tasks lead to less compact and interpretable code explanations (**Q3**). We quantify compactness and interpretability using four metrics: (i) number of lines, (ii) number of characters, (iii) Pylint score [24], which measures code quality and readability, and (iv) cyclomatic complexity [30], which measures the branching complexity of a program. We provide further details on each metric in Section B. Figure 3 reports the average metric values across selected code explanations for each task, with each metric normalized using z-scores to enable comparison across scales. We find no systematic difference in the number of lines and Pylint score between the first and the last four tasks, whereas the last four, more complex tasks result in code explanations with slightly more characters and higher cyclomatic complexity. These results suggest that more complex tasks lead to slightly more complex and denser code explanations with more code per line, while still adhering to standard Python coding conventions.

To further examine this trend, we manually inspect the most consistent code explanation in each of the 24 model-task settings, with examples provided in Section D and Table 2. All inspected programs rely on linguistic pattern matching to some extent. The number and length of hard-coded patterns increase with the linguistic complexity of the task. For simpler tasks, pattern matching is limited to a small set of rules, whereas for more complex tasks, programs hard-code a larger number of longer words and phrases. This suggests that, for tasks with complex linguistic components, LLMs explain their behavior by stitching together case-specific patterns rather than writing compact, general procedures. Overall, as linguistic complexity increases, LLMs produce longer and structurally more complex code explanations that rely more heavily on hard-coded patterns, making them less interpretable to humans. This provides an answer to **(Q3)**.

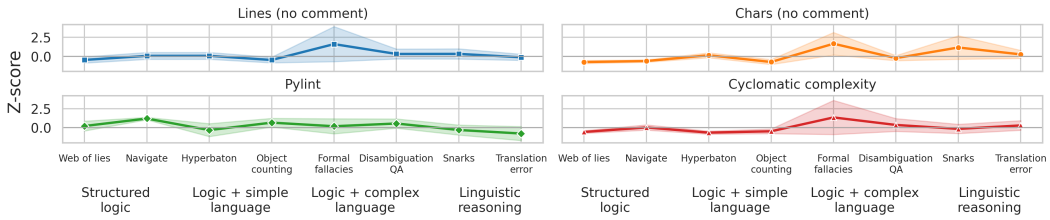


Figure 3: Measures of compactness and interpretability for selected code explanations, averaged across models and seeds, with standard deviation shown. More complex tasks lead to code explanations with slightly longer lines and more complex branching structure.

Table 2: Example snippets of code explanations for tasks across 4 categories. The extent of hard-coding increases with the linguistic complexity of the task.

Category	Task	Code snippet

Structured logic	Navigate	`if "forward" in words:`
		`  if direction == 0:`
		`  y_pos += steps`

Logic + simple language	Hyperbaton	`def categorize_adjective(adj):`
		`  if adj in ["awful", "repulsive", "obnoxious", "terrible", "good", "nice", "lovely"]:`
		`  return "opinion"`

Logic + complex language	Formal fallacies	`premises = [premise.replace(' is ', '_is_').replace(' or ', '_or_').replace(' and ', '_and_').replace(' every ', 'every_').replace(' whoever ', 'whoever_').replace(' being a ', 'being_a_').replace(' an ', 'an_').replace(' a ', 'a_') for premise in premises]` `premises = [premise.replace('not ', 'not_') for premise in premises]`
Linguistic reasoning	Snarks	`if any(indicator in statement for indicator in ['ARE U INSANE', 'Have my down vote', 'naive', 'terrible support', 'pragmatic', 'more expensive', 'less fun', 'liability', 'enjoyable', 'best', 'reddit', 'fun', 'sexist', 'win you points', 'lose you points', 'True', 'rational', 'threaten', 'pleaded', 'highest', 'lowest', 'rich', 'loans', 'suspicious', 'legit', 'perfect']):` ` sarcastic_option = label`

5.3 Compare code with free-text explanations

We investigate whether the theoretical advantages of code explanations translate into more faithful and compact explanations than free-text explanations generated by current LLMs. To evaluate CoT, we use SCSuff [33], which measures how well an explanation captures model behavior by comparing the model's output distributions on the original inputs with its distributions on possible alternative inputs that the model considers possible given the explanation, using KL divergence. To enable direct comparison between CoT and code explanations, we adapt our consistency metric to use the expected KL divergence from SCSuff, assigning 99% probability to the answer returned by the code explanation. We measure the compactness of the two explanation types using their token counts. Further details on the evaluation procedure are provided in Section B.

The top panel of Figure 4 shows that neither explanation type consistently achieves higher sufficiency. Code explanations achieve higher sufficiency in 9 out of 24 settings, split between simpler (5 settings) and more complex tasks (4 settings). Thus, our code explanation generation procedure can produce explanations that are more sufficient than per-instance CoT. However, on more complex tasks, code explanations are less often more sufficient than CoT, which can provide a tailored explanation for each input. At the same time, the bottom panel of Figure 4 shows that code explanations are $6\times$ to $100\times$ more compact than CoT across all tasks. This is expected, as the total length of per-instance CoT explanations grows with the size of the explained dataset. Thus, although code explanations do not consistently achieve higher sufficiency than free-text CoT with our one-shot prompting approach, they already offer a substantial advantage in compactness.

5.4 Can code explanation consistency be improved?

Privileged self-access. Prior work suggests that LLMs encode world knowledge [36, 43] and prediction uncertainty [38] in their internal representations. If code consistency is also encoded in these representations, privileged self-access [4, 39] could help models generate more consistent code

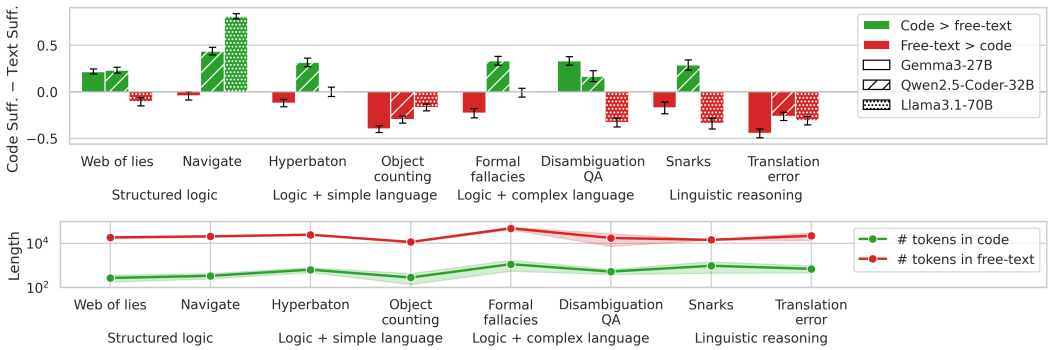


Figure 4: **(Top)** Differences in sufficiency between code and free-text CoT explanations across models and tasks, with 95% CI of the average sample-level differences. Neither code nor free-text consistently achieves higher sufficiency across all settings. **(Bottom)** Token counts for code and free-text CoT explanations over the test dataset for each task, averaged across models and seeds, with standard deviation shown. Code explanations are 6× to 100× shorter than CoT across settings.

explanations. We therefore test whether these representations can predict code explanation consistency before generation. Figure 8 visualizes final-token hidden states from 10 code-generation prompts per model-task setting, colored by the best-of-10 consistency of the generated code. The visualization shows no clear separation by consistency, and ridge regression yields negative 5-fold cross-validation R^2 across all models when predicting either CONSISTENCY or ΔCONSISTENCY. These results suggest that code consistency is not readily predictable from LLMs' internal representations, limiting LLMs' ability to improve explanation consistency through internal representations alone.

Privileged self-access may not be necessary to obtain consistent explanations. To test this, Figure 5 compares the code consistency of Gemma and Qwen when explaining their own versus each other's outputs, using one random seed. In 12 of 16 model-task settings, self-explanations and cross-model explanations achieve consistency within 0.08 of each other. In 3 of the remaining 4 settings, cross-model explanations achieve higher consistency. These results suggest that consistency feedback can identify post-hoc explanations that best match a target model's behavior, even when generated by a different LLM.

Verification feedback. We next investigate whether execution and consistency feedback can improve code explanation consistency in multi-turn agentic settings. We compare four approaches in Figure 6 using one random seed: (1) the best one-shot model among the three previously studied; (2) 10-turn iterative refinement with the same models, with the most consistent program generated so far and its consistency provided after each turn; (3) Claude Opus 5.5 with Python execution; and (4) Claude Opus 5.5 with both Python execution and a consistency-evaluation tool. See Section B for the full implementation details.

Figure 6 shows that multi-turn refinement improves consistency on the first four, simpler tasks. Figure 9 further plots code consistency across 10 refinement turns for each model-task setting. On the first four, simpler tasks, ΔCONSISTENCY approaches the maximum in all but two settings. In contrast, ΔCONSISTENCY remains close to zero across turns on the last four, more complex tasks. These results suggest that multi-turn refinement can improve consistency on simpler tasks, but is still insufficient to overcome the challenges of generating consistent explanations for more complex tasks.

Both Claude variants achieve the highest ΔCONSISTENCY (Figure 6) while producing code of comparable complexity to open-source models (Figure 11), highlighting the benefit of execution feedback, with or without the consistency-evaluation tool. We provide examples of Claude-generated code in Section D. Figure 10 further compares Claude's task accuracy and explanation consistency. On the first four tasks, Claude achieves 100% on both metrics, showing that perfectly consistent explanations are possible, although perfect task accuracy makes it impossible to distinguish consistent explanation from accurate task solving. On the last four tasks, Claude maintains task accuracy above 75% but achieves ΔCONSISTENCY ≤ 0.3. These improvements nevertheless exceed the uninformative majority-answer baseline, showing that execution feedback can improve code consistency across all tasks, although consistency remains substantially lower on more complex tasks.

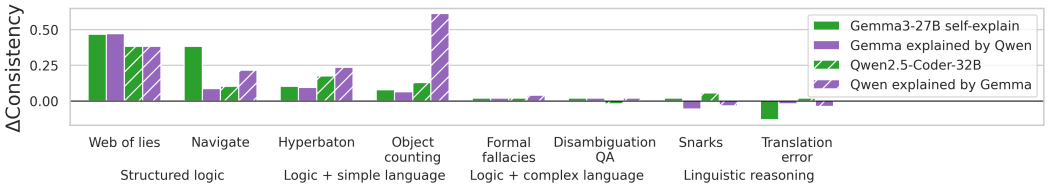


Figure 5: Δ CONSISTENCY of self-explanations and cross-model explanations, where Gemma and Qwen explain their own behavior or each other's, respectively. In 12 of the 16 cases, the two approaches achieve consistency within 0.08 of each other, showing that consistency feedback can benefit both self-explanations and post-hoc explanations.

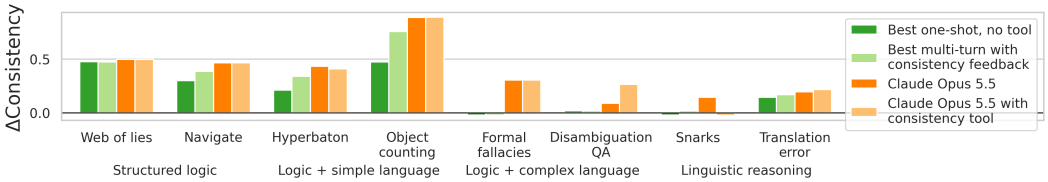


Figure 6: Δ CONSISTENCY across four approaches: (1) one-shot generation, (2) 10-turn iterative refinement with consistency feedback, (3) Claude with code execution, and (4) Claude with execution and consistency feedback. Multi-turn refinement and agentic execution with Claude Opus 5.5 improve consistency over one-shot generation, though consistency remains substantially lower on the last four, more complex tasks.

6 Conclusion

We propose executable code as an automatically verifiable and compact alternative to instance-wise free-text explanations of LLM behavior. We introduce CONSISTENCY, a metric that measures how often an executable explanation reproduces the model's outputs across inputs, and use it to evaluate and select candidate explanations. Experiments across three code-capable LLMs and eight tasks show that highly consistent code explanations are possible in several settings. Code explanations better capture model behavior than free-text CoT in 9 out of 24 settings, while being $6\times$ to $100\times$ shorter. However, high consistency is not the norm. On tasks with more complex linguistic features, code explanations often perform near random chance and tend to rely on longer hard-coded linguistic patterns. Multi-turn refinement improves consistency on simpler tasks, but yields no improvement on more complex ones. Experiments with Claude Opus 5.5 show that equipping an LLM agent with code execution and consistency feedback improves consistency across all tasks, but even Claude struggles to achieve high consistency on more complex tasks. Together, these findings highlight both the advantages of executable explanations as a verifiable and compact approach to explaining LLM behavior and the challenge of reliably producing consistent explanations, motivating future optimization through execution and consistency feedback.

Limitations. Our evaluation is limited to Python code explanations and fixed generation hyperparameters, which may affect the expressivity, compactness, and consistency of code explanations. We leave exploring these extensions and systematically optimizing generation hyperparameters for future work. Further, our metric measures behavioral consistency but does not establish whether code explanations faithfully describe the model's internal computations.

Interpreting low consistency on linguistically complex tasks. The low consistency on linguistically complex tasks in our experiments has two possible interpretations. One is that current LLMs lack the ability to generate code that accurately captures their own behavior on tasks involving complex linguistic features. The other is that some linguistic behaviors are inherently difficult to represent through compact, human-interpretable programs, even with more capable explanation methods. Distinguishing between these possibilities is an important direction for future work, both for improving executable explanations and for understanding the limits of compact, human-interpretable representations of LLM behavior.

Acknowledgments

This work was partly supported by the NIH/NHLBI Award R01HL148248, NSF Award 1922658 NRT-HDR: FUTURE Foundations, Translation, and Responsibility for Data Science, NSF CAREER Award 2145542, ONR N00014-23-1-2634, NIH R01CA296388, NSF 2404476, Optum, the Gates Foundation, and Apple. This work was also supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) with a grant funded by the Ministry of Science and ICT (MSIT) of the Republic of Korea in connection with the Global AI Frontier Lab International Collaborative Research Agreement. (No. RS-2024-00469482 & RS-2024-00509279).

References

[1] Anthropic. Claude Opus 5.5, 2026. URL `https://platform.claude.com/docs/en/models/opus-5-5/overview`.

[2] Pepa Atanasova, Oana-Maria Camburu, Christina Lioma, Thomas Lukasiewicz, Jakob Grue Simonsen, and Isabelle Augenstein. Faithfulness tests for natural language explanations. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 283–294, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-short.25.

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

[4] Felix Jedidja Binder, James Chua, Tomek Korbak, Henry Sleight, John Hughes, Robert Long, Ethan Perez, Miles Turpin, and Owain Evans. Looking inward: Language models can learn about themselves by introspection. In *International Conference on Learning Representations*, volume 2025, pages 3710–3756, 2025.

[5] Jianbo Chen, Le Song, Martin Wainwright, and Michael Jordan. Learning to explain: An information-theoretic perspective on model interpretation. In *International conference on machine learning*, pages 883–892. PMLR, 2018.

[6] Junying Chen, Chi Gui, Anningzhe Gao, Ke Ji, Xidong Wang, Xiang Wan, and Benyou Wang. Cod, towards an interpretable medical agent using chain of diagnosis. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 14345–14368, 2025.

[7] 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. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Yanda Chen, Joe Benton, Ansh Radhakrishnan, Jonathan Uesato, Carson Denison, John Schulman, Arushi Somani, Peter Hase, Misha Wagner, Fabien Roger, Vlad Mikulik, Samuel Bowman, Jan Leike, Jared Kaplan, and Ethan Perez. Reasoning models don't always say what they think. *arXiv preprint arXiv:2505.05410*, 2025.

[9] Barry De Ville. Decision trees. *Wiley Interdisciplinary Reviews: Computational Statistics*, 5(6): 448–455, 2013.

[10] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

[11] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021.

[13] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[14] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025.

[15] Neil Jethani, Mukund Sudarshan, Yindalon Aphinyanaphongs, and Rajesh Ranganath. Have we learned to explain?: How interpretability methods can learn to encode predictions in their interpretations. In *International Conference on Artificial Intelligence and Statistics*, pages 1459–1467. PMLR, 2021.

[16] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *ACM Transactions on Software Engineering and Methodology*, 35 (2):1–72, 2026.

[17] Lavender Yao Jiang, Xujin Chris Liu, Nima Pour Nejatian, Mustafa Nasir-Moin, Duo Wang, Anas Abidin, Kevin Eaton, Howard Antony Riina, Ilya Laufer, Paawan Punjabi, et al. Health system-scale language models are all-purpose prediction engines. *Nature*, 619(7969):357–362, 2023.

[18] Been Kim, John Hewitt, Neel Nanda, Noah Fiedel, and Oyvind Tafjord. Because we have llms, we can and should pursue agentic interpretability. *arXiv preprint arXiv:2506.12152*, 2025.

[19] Dongki Kim, Wonbin Lee, and Sung Ju Hwang. Mol-llama: Towards general understanding of molecules in large molecular language model. In *Advances in Neural Information Processing Systems*, volume 38, pages 26921–26960, 2026.

[20] Satyapriya Krishna, Jiaqi Ma, Dylan Slack, Asma Ghandeharioun, Sameer Singh, and Himabindu Lakkaraju. Post hoc explanations of language models can improve language models. In *Advances in Neural Information Processing Systems*, volume 36, pages 65468–65483, 2023.

[21] Nicholas Kroeger, Dan Ley, Satyapriya Krishna, Chirag Agarwal, and Himabindu Lakkaraju. Are large language models post hoc explainers? In *XAI in Action: Past, Present, and Future Applications*, 2023.

[22] Michele Lacchia. Radon, 2023. URL `https://github.com/rubik/radon`.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 2002.

[24] Logilab. Pylint, 2026. URL `https://github.com/pylint-dev/pylint`.

[25] Ziyang Ma, Zhuo Chen, Yuping Wang, Eng-Siong Chng, and Xie Chen. Audio-cot: Exploring chain-of-thought reasoning in large audio language model. In *2025 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, pages 1–6. IEEE, 2025.

[26] Andreas Madsen, Sarath Chandar, and Siva Reddy. Are self-explanations from large language models faithful? In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 295–337, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.19.

[27] Samuel Marks, Can Rager, Eric Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. In *International Conference on Learning Representations*, volume 2025, pages 23888–23923, 2025.

[28] Katie Matton, Robert Ness, John Guttag, and Emre Kiciman. Walk the talk? measuring the faithfulness of large language model explanations. In *The Thirteenth International Conference on Learning Representations*, 2025.

[29] Harry Mayne, Ryan Othniel Kearns, Yushi Yang, Andrew M Bean, Eoin D Delaney, Chris Russell, and Adam Mahdi. Llms don't know their own decision boundaries: The unreliability of self-generated counterfactual explanations. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 24172–24197, 2025.

[30] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

[31] Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations*, 2023.

[32] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 135(3):370–384, 1972.

[33] Nhi Nguyen, Shauli Ravfogel, and Rajesh Ranganath. What llms explain is not what they believe: Evaluating explanation sufficiency under models' own input beliefs. In *International conference on machine learning*, 2026.

[34] OpenAI. The hugging face incident and the road ahead. `https://openai.com/index/hugging-face-incident-and-the-road-ahead/`, 2026.

[35] Letitia Parcalabescu and Anette Frank. On measuring faithfulness or self-consistency of natural language explanations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6048–6089, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.329.

[36] Core Francisco Park, Andrew Lee, Ekdeep Singh Lubana, Yongyi Yang, Maya Okawa, Kento Nishi, Martin Wattenberg, and Hidenori Tanaka. Iclr: In-context learning of representations. In *The Thirteenth International Conference on Learning Representations*, 2025.

[37] Eric A Posner and Shivam Saran. Judge ai: A case-study of large language models as judges. *Journal of Law & Empirical Analysis*, 3(1):179–203, 2026.

[38] Aviv Slobodkin, Omer Goldman, Avi Caciularu, Ido Dagan, and Shauli Ravfogel. The curious case of hallucinatory (un)answerability: Finding truths in the hidden states of over-confident large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3607–3625, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.220.

[39] Siyuan Song, Jennifer Hu, and Kyle Mahowald. Language models fail to introspect about their knowledge of language. *arXiv preprint arXiv:2503.07513*, 2025.

[40] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051, 2023.

[41] Tianyi Tang, Wenyang Luo, Haoyang Huang, Dongdong Zhang, Xiaolei Wang, Wayne Xin Zhao, Furu Wei, and Ji-Rong Wen. Language-specific neurons: The key to multilingual capabilities in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5701–5715, 2024.

[42] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[43] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet. *Transformer Circuits Thread*, 2024. URL `https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html`.

[44] Kunvar Thaman. Reward hacking benchmark: Measuring exploits in llm agents with tool use. In *International conference on machine learning*, 2026.

[45] Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems*, volume 36, pages 74952–74965, 2023.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, volume 30, 2017.

[47] Hao Wang, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. How we broke top ai agent benchmarks: And what comes next. *Berkeley RDI Blog, April*, 2026.

[48] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 50208–50232. PMLR, 21–27 Jul 2024.

[49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in neural information processing systems*, volume 35, pages 24824–24837, 2022.

[50] Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh Agrawal, Sandeep Sandha, Siddartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-limited llm benchmark. In *International Conference on Learning Representations*, volume 2025, pages 91595–91631, 2025.

[51] Ke Yang, Jiateng Liu, John Wu, Chaoqi Yang, Yi R Fung, Sha Li, Zixuan Huang, Xu Cao, Xingyao Wang, Heng Ji, and ChengXiang Zhai. If llm is the wizard, then code is the wand: A survey on how code empowers large language models to serve as intelligent agents. In *International conference on learning representations*, 2027.

[52] Jinsung Yoon, James Jordon, and Mihaela Van der Schaar. Invase: Instance-wise variable selection using neural networks. In *International conference on learning representations*, 2018.

[53] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[54] Xuan Zhang, Chao Du, Tianyu Pang, Qian Liu, Wei Gao, and Min Lin. Chain of preference optimization: Improving chain-of-thought reasoning in llms. In *Advances in Neural Information Processing Systems*, volume 37, pages 333–356, 2024.

Appendix

A Figures

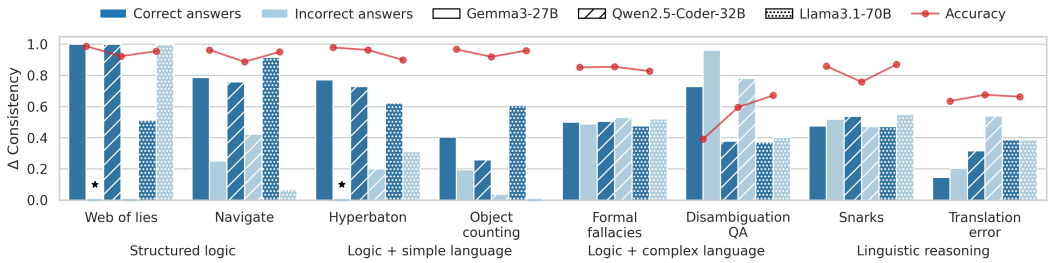


Figure 7: CONSISTENCY stratified by model answer correctness. In 13/24 cases, consistency is no more than $+0.1$ higher for correct than incorrect answers, and in 10/24 cases consistency is higher for incorrect answers, suggesting that code explanations do not systematically explain only the model's correct answer or simply follow the task instruction. In 4 cases, consistency on incorrect answers is 0, but all achieve accuracy $\geq 92\%$, suggesting that imperfectly consistent programs may capture the model's behavior on most examples while failing to capture its occasional mistakes. $\star$ indicate averages computed from fewer than 10 examples, which occur when accuracy is near 100% and few incorrect answers are available.

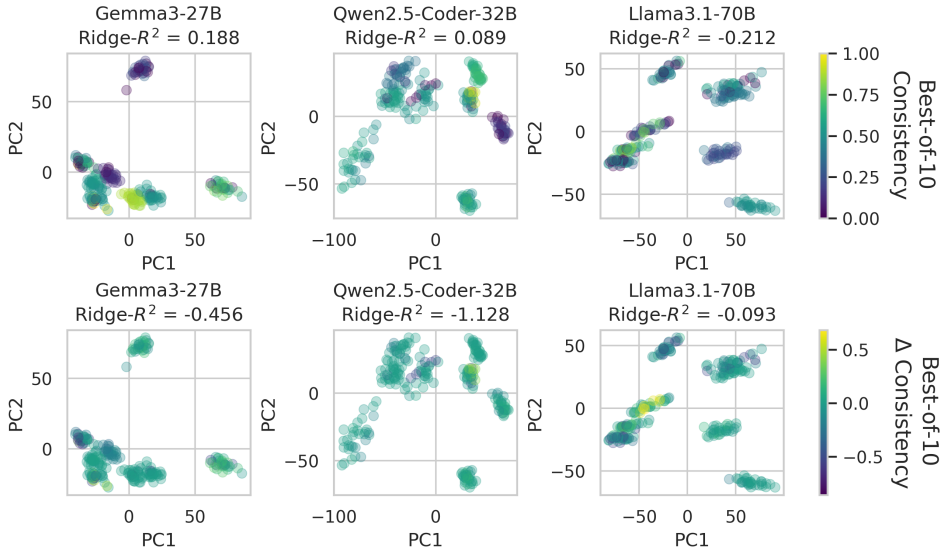


Figure 8: PCA visualization of the last hidden state of the final input token across LLMs. Each sample corresponds to one random seed and one set of $N = 20$ in-context examples among the $P = 10$ code-generation prompts for a model-task setting. The samples are colored by their best-of-M ($M = 10$) code CONSISTENCY (top row) and ΔCONSISTENCY (bottom row). Each panel reports the 5-fold cross-validation R^2 for a ridge regression model trained to predict best-of-10 CONSISTENCY (ΔCONSISTENCY resp.) from the hidden states. Across all models, ridge regression results in negative R^2, and the hidden states show no clear separation by either consistency measure, suggesting that LLM hidden states do not encode information about whether LLMs will produce consistent explanations.

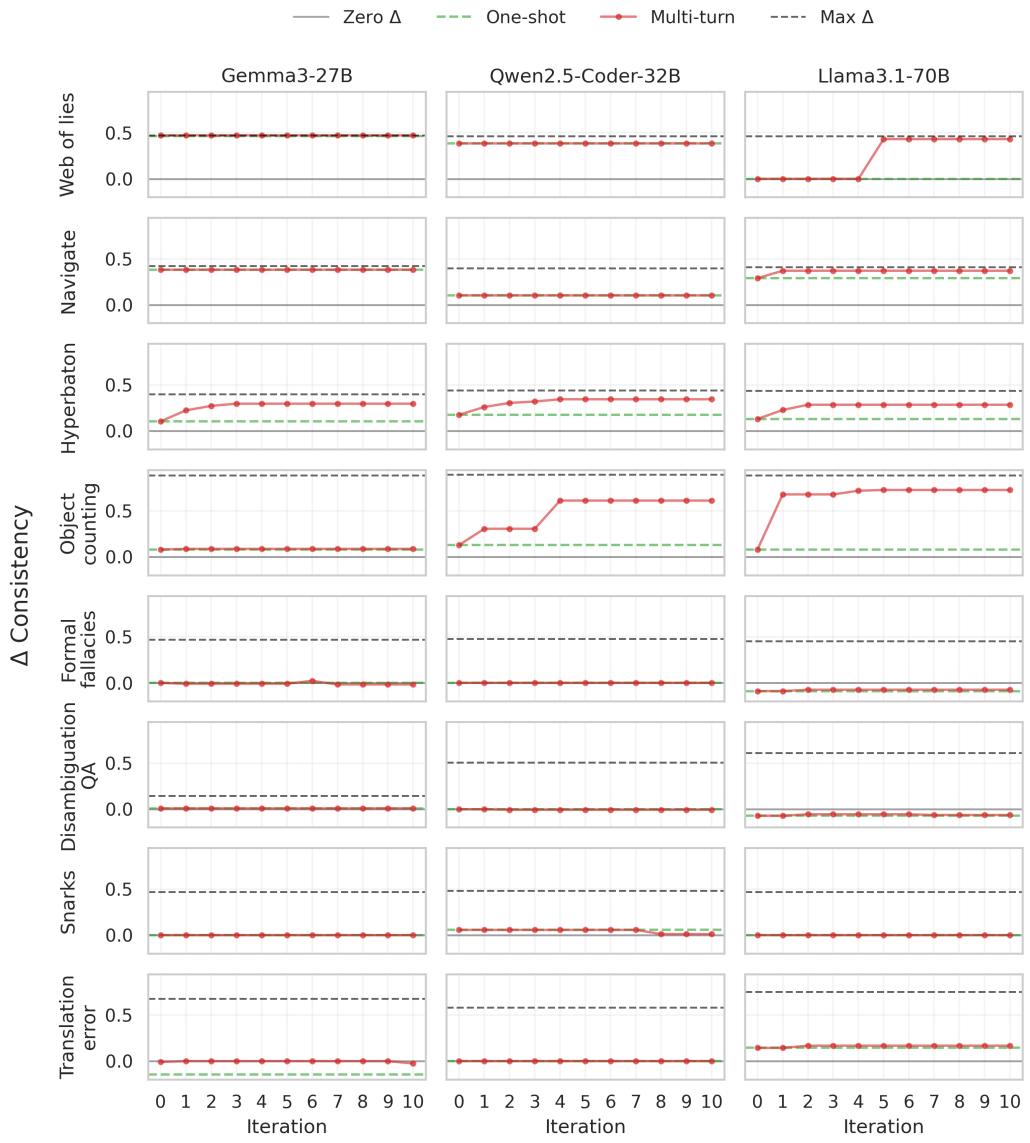


Figure 9: Code consistency across 10 refinement turns for each model (column) and task (row) pair. The zero and maximum ΔCONSISTENCY lines are annotated for each setting. On the first four tasks, ΔCONSISTENCY approaches its maximum after 10 turns in all but two settings. On the last four tasks, code consistency remains near its one-shot level, with no improvement across turns.

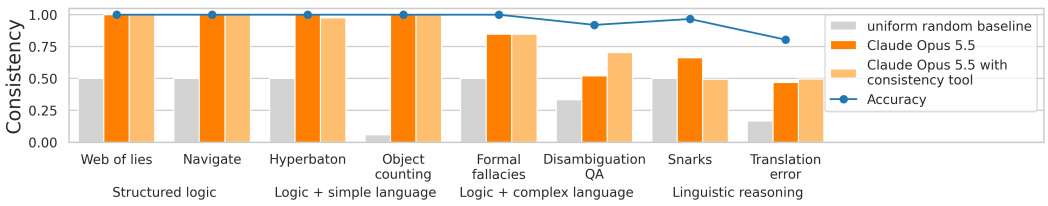


Figure 10: Task accuracy and code consistency of Claude, with and without an explicit consistency-evaluation tool, across eight tasks. Both variants achieve close to 100% on both metrics for the first four tasks. On the last four, accuracy remains above 75%, but code consistency is at most +0.3 higher than the uniform-random baseline.

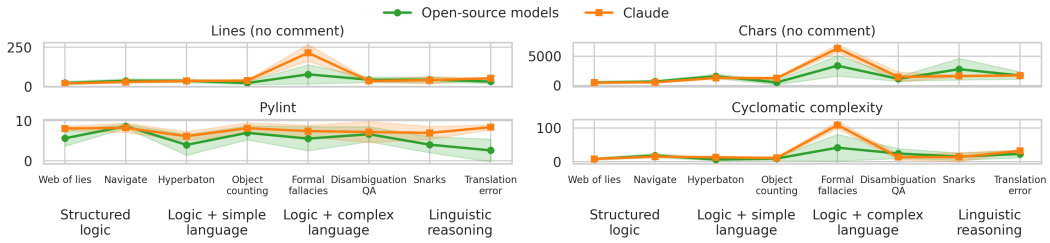


Figure 11: Measures of compactness and interpretability of the best code explanations generated by open-source models and Claude, with standard deviations shown. Across tasks, the complexity of code explanations is similar between open-source models and Claude. Only for the Formal Fallacies task, all models generate explanations with significantly more lines, characters, and higher cyclomatic complexity.

B Implementation details

Table 3: Prompt used to generate code explanations for LLM's behavior on a dataset.

Role	Content
System	`You are given a task along with N example input-output pairs that you previously answered. Your task is to explain your underlying algorithm. The function must correctly handle all tokens that appear in the inputs. Do not ignore any part of the input string.`
User	`Task description: T; Q: x_1, A: y_1; ... ; Q: x_N, A: y_N` `Write a Python function that reproduces the outputs from the inputs. Think about how to write an algorithm step by step. Make sure that your code does not raise any error when running with the input strings. Do not show example usage. Output the final Python code in the following format:`

Generate answers. Model answers are generated with chain-of-thought prompting, temperature 0, repetition penalty 1.0, and a maximum of 2000 new tokens.

Generate code explanations. Code explanations are also generated with chain-of-thought prompting, but with temperature 0.1, top-p 0.95, repetition penalty 1.0, and a maximum of 20000 new tokens. We use Table 3 to generate code explanations in Python in the format `solve(x)`, where `x` is the full input text. Although most generated programs follow the `solve(x)` interface, a small number of cases (Translation Error and Disambiguation QA) produce two-argument functions (e.g., `solve(source, translation)`); we handle these edge cases with an input processing step to enable evaluation. For task-instruction-only experiments, we remove the model's answers from the prompt and select code explanations by their consistency with ground-truth answers on the training set.

Measure compactness and interpretability. To measure compactness and interpretability of selected code explanations, we measure the number of lines after removing empty lines and comment-only lines, and the number of characters after removing all comments. We compute the Pylint score using the `pylint` library [24], which ranges from 0 to 10 and measures how well Python code conforms to coding standards, including readability, error-proneness, and stylistic conventions. We compute cyclomatic complexity using the `radon` library [22], which measures the complexity of a program by computing the number of linearly independent paths through a source code.

Measure free-text explanation sufficiency. Self-Consistent Sufficiency [33] evaluates how well an explanation $\boldsymbol{e}$ explains an LLM's behavior on an input $\boldsymbol{x}$ by computing the KL divergence between the model's output distribution given $\boldsymbol{x}$ and the average model's output distributions over alternative inputs $\boldsymbol{x}'$ that the LLM itself considers possible given $\boldsymbol{e}$:

$$\mathcal{S}(\boldsymbol{x}, \boldsymbol{e}, F) := \mathrm{KL}\left[F_{\mathbf{y}\,|\,\mathbf{x}}(\cdot\,|\,\boldsymbol{x}) \middle\| \mathbb{E}_{\mathbf{x}'\,|\,\mathbf{e}}\left[F_{\mathbf{y}\,|\,\mathbf{x}}(\cdot\,|\,\boldsymbol{x}')\right]\right] \tag{5}$$

We compute free-text sufficiency scores using the same code and hyperparameters as Nguyen et al. [33]. Given a code explanation c, we evaluate its sufficiency by replacing $\mathbb{E}_{\mathbf{x}' \mid \mathbf{e}}\left[F_{\mathbf{y} \mid \mathbf{x}}(\cdot \mid \boldsymbol{x}')\right]$ in Equation (5) with an output distribution that assigns 99% probability to $c(\boldsymbol{x})$ and distributes the remaining probability equally among other valid output options. For code explanations, token length is the number of tokens in the generated program. For CoT explanations, token length is the total number of tokens across all CoTs generated for the test set, with one CoT for each input. We evaluated CoT and free-text explanations on the same test split as the main consistency experiments.

Predict CONSISTENCY from hidden states. For each model-task setting, we extract 80 hidden-state vectors, corresponding to the $P = 10$ code-generation prompts for each of the eight tasks. For each prompt, we compute the best-of-M ($M = 10$) consistency among its generated programs. To predict best-of-10 CONSISTENCY and ΔCONSISTENCY using LLM hidden states, we use standard 5-fold cross-validation with random fold assignment, standardize features using `StandardScaler`, and fit a ridge regression model with $\alpha = 1.0$ using `scikit-learn` library.

Multi-turn refinements. We use the same initial prompts as in one-shot generation and perform 10 refinement turns. At each turn, we append the best code explanation generated so far and its code consistency score on the training set to the original explanation prompt, and further instruct the model to improve consistency. We evaluate the resulting code explanations on the test set after each turn. Multi-turn refinement experiments are done using one random seed.

Claude agent. We use Claude Opus 5.5 with access to Python execution and file-editing tools, allowing it to write, execute, test, and revise code explanations. Each agent run is limited to 10 turns and a $1 budget. In the consistency-feedback variant, we additionally provide a tool that computes code consistency on the training set using the same evaluation function as in our experiments. The agent can use this score to revise its program. We evaluate the final programs on the test set. Claude code experiments are done using one random seed.

Resources. Computing all metrics on each model-task setting on a single seed takes 2-10 hours on 2-4 A100 GPUs.

C Task descriptions and examples

We provide a representative example question from each of the eight BBH [40] tasks used in our experiments.

Web of lies. Question: Sherrie tells the truth. Vernell says Sherrie tells the truth. Alexis says Vernell lies. Michaela says Alexis tells the truth. Elanor says Michaela tells the truth. Does Elanor tell the truth?

Navigate. If you follow these instructions, do you return to the starting point? Always face forward. Take 1 step backward. Take 9 steps left. Take 2 steps backward. Take 6 steps forward. Take 4 steps forward. Take 4 steps backward. Take 3 steps right.
Options:
- Yes
- No

Hyperbaton. Which sentence has the correct adjective order:
Options:
(A) midsize old grey Brazilian sweater
(B) midsize grey Brazilian old sweater

Object counting. I have a flute, a piano, a trombone, four stoves, a violin, an accordion, a clarinet, a drum, two lamps, and a trumpet. How many musical instruments do I have?

Formal fallacies. "Here comes a perfectly valid argument: First of all, whoever is a schoolmate of Sondra is not a stepsister of Pricilla. In consequence, whoever is not a stepsister of Pricilla is a schoolmate of Sondra." Is the argument, given the explicitly stated premises, deductively valid or invalid?

Options:
- valid
- invalid

Disambiguation QA. In the following sentences, explain the antecedent of the pronoun (which thing the pronoun refers to), or state that it is ambiguous.
Sentence: The patient was referred to the specialist because he had a rare skin condition.
Options:
(A) The patient had a skin condition
(B) The specialist had a skin condition
(C) Ambiguous

Snarks. Which statement is sarcastic?
Options:
(A) He's a generous person, trying to promote a charity stream that has raised millions to help kids in need
(B) He's a terrible person, trying to promote a charity stream that has raised millions to help kids in need

Translation error. The following translations from German to English contain a particular error. That error will be one of the following types: Named Entities: An entity (names, places, locations, etc.) is changed to a different entity. Numerical Values: Numerical values (ordinals or cardinals), dates, and/or units are changed. Modifiers or Adjectives: The modifiers and adjectives pertaining to a noun are changed. Negation or Antonyms: Introduce or remove a negation or change comparatives to their antonyms. Facts: Trivial factual errors not pertaining to the above classes are introduced in the translations. Dropped Content: A significant clause in the translation is removed. Please identify that error. Source: Karl Borromäus Joseph Fürst von Liechtenstein war ein kaiserlicher Feldmarschall.
Translation: Charles Borromeo Joseph Prince of Liechtenstein was an judicial field marshal.
The translation contains an error pertaining to
Options:
(A) Modifiers or Adjectives
(B) Numerical Values
(C) Negation or Antonyms
(D) Named Entities
(E) Dropped Content
(F) Facts

D Code explanation with highest ΔCONSISTENCY for each task

We provide the full code explanation with the highest ΔCONSISTENCY on the test set from the open-source models for each task, alongside the higher-performing of the two Claude variants.

Web of lies. Gemma3-27B

```python
def solve(x):
    """
    This function determines if the last person in a chain of statements tells the truth.
    The chain consists of people making statements about the truthfulness of the previous person.
    The function assumes that exactly one person in the chain is lying.
    """
    statements = x.split(". ")
    n = len(statements)

    # Initialize a list to store the truthfulness of each person.
    # True means the person tells the truth, False means they lie.
    truth_values = [True] * n

    # Iterate through the statements and update the truth values based on the statements.
    for i in range(n - 1):
        statement = statements[i]
        name = statement.split(" says ")[0]

        if "lies" in statement:
            truth_values[i+1] = not truth_values[i]
        else:
            truth_values[i+1] = truth_values[i]
```

```
    # Check if the last person tells the truth.
    if truth_values[n-1]:
        return "Yes"
    else:
        return "No"
```

Claude Opus 5.5

```
import re
def solve(x):
    s = x.split("Question:")[-1]
    truth = {}
    m = re.search(r"(\w+) (tells the truth|lies)\.", s)
    truth[m.group(1)] = m.group(2) == "tells the truth"
    for a, b, c in re.findall(r"(\w+) says (\w+) (tells the truth|lies)", s):
        truth[a] = truth.get(b, True) == (c == "tells the truth")
    q = re.search(r"Does (\w+) tell the truth", s)
    return "Yes" if truth.get(q.group(1), False) else "No"
```

Navigate. Gemma3-27B

```
def solve(x):
    instructions = x.lower().split('.')
    x_pos = 0
    y_pos = 0
    direction = 0  # 0: forward, 1: right, 2: backward, 3: left

    for instruction in instructions:
        instruction = instruction.strip()
        if not instruction:
            continue

        words = instruction.split()

        if "take" in words:
            try:
                steps = int(words[words.index("take") + 1])
            except (ValueError, IndexError):
                continue

            if "forward" in words:
                if direction == 0:
                    y_pos += steps
                elif direction == 2:
                    y_pos -= steps
                elif direction == 1:
                    x_pos += steps
                elif direction == 3:
                    x_pos -= steps
            elif "backward" in words:
                if direction == 0:
                    y_pos -= steps
                elif direction == 2:
                    y_pos += steps
                elif direction == 1:
                    x_pos -= steps
                elif direction == 3:
                    x_pos += steps
            elif "right" in words:
                x_pos += steps
            elif "left" in words:
                x_pos -= steps
            else:
                if direction == 0:
                    y_pos += steps
                elif direction == 2:
                    y_pos -= steps
                elif direction == 1:
                    x_pos += steps
                elif direction == 3:
                    x_pos -= steps

        elif "turn" in words:
            if "right" in words:
                direction = (direction + 1) % 4
            elif "left" in words:
                direction = (direction - 1) % 4
            elif "around" in words:
                direction = (direction + 2) % 4

        elif "always face forward" in words:
```

```
            pass

    if x_pos == 0 and y_pos == 0:
        return "Yes"
    else:
        return "No"
```

Claude Opus 5.5

```
import re
def solve(x):
    s = x.split('?', 1)[1] if '?' in x else x
    s = s.split('Options:')[0]
    x0 = y0 = 0
    dx, dy = 0, 1
    for ins in [t.strip() for t in s.split('.') if t.strip()]:
        low = ins.lower()
        if low.startswith('turn'):
            if 'around' in low: dx, dy = -dx, -dy
            elif 'left' in low: dx, dy = -dy, dx
            elif 'right' in low: dx, dy = dy, -dx
        elif low.startswith('take'):
            m = re.search(r'\d+', low)
            n = int(m.group()) if m else 0
            if 'backward' in low: vx, vy = -dx, -dy
            elif 'left' in low: vx, vy = -dy, dx
            elif 'right' in low: vx, vy = dy, -dx
            else: vx, vy = dx, dy
            x0 += vx*n; y0 += vy*n
    return 'Yes' if x0 == 0 and y0 == 0 else 'No'
```

Hyperbaton. Gemma3-27B

```
def solve(x):
    """
    This function determines the sentence with the correct adjective order.

    The adjective order is generally as follows:
    1. Opinion
    2. Size
    3. Physical Quality (shape, age, color)
    4. Origin
    5. Material
    6. Type
    7. Purpose

    The function compares the adjective order in the two given sentences and returns the letter of the
    ↪  sentence with the correct order.
    """

    adj_order = ["opinion", "size", "physical_quality", "age", "shape", "color", "origin", "material",
    ↪  "type", "purpose"]

    def categorize_adjective(adj):
        adj = adj.lower()
        if adj in ["awful", "repulsive", "obnoxious", "terrible", "good", "nice", "lovely"]:
            return "opinion"
        elif adj in ["huge", "big", "small", "medium-size", "normal-size", "massive"]:
            return "size"
        elif adj in ["circular", "square", "triangular", "prismlike", "rectangular"]:
            return "shape"
        elif adj in ["old", "ancient", "old-fashioned", "brand-new", "archaic"]:
            return "age"
        elif adj in ["red", "yellow", "purple", "green", "tan", "silver", "gold", "orange"]:
            return "color"
        elif adj in ["american", "japanese", "pakistani", "iranian", "filipino", "vietnamese",
        ↪  "indonesian"]:
            return "origin"
        elif adj in ["wool", "fiberglass", "wood", "glass"]:
            return "material"
        elif adj in ["exercise", "drinking", "eating", "walking", "smoking", "hiking"]:
            return "purpose"
        else:
            return "type"  # Default to type if not found

    def get_adjective_categories(sentence):
        words = sentence.split()
        adjectives = []
        for word in words:
            category = categorize_adjective(word)
            if category != "type":
```

```
                adjectives.append((word, category))
        return adjectives

    sentence1 = x.split("(A) ")[1].split("\n")[0]
    sentence2 = x.split("(B) ")[1].split("\n")[0]

    adjectives1 = get_adjective_categories(sentence1)
    adjectives2 = get_adjective_categories(sentence2)

    def check_order(adjectives):
        order = [adj[1] for adj in adjectives]
        for i in range(len(order) - 1):
            if adj_order.index(order[i]) > adj_order.index(order[i+1]):
                return False
        return True

    if check_order(adjectives1):
        return "A"
    else:
        return "B"
```

Claude Opus 5.5

```
import re

CATS = {
 1: "nice lovely repulsive silly terrible awful ridiculous mysterious obnoxious good bad wonderful
 ↪  beautiful ugly adorable grotesque enormous-looking fantastic horrible",
 2: "little big small tiny massive midsize medium-size large huge enormous giant",
 3: "brand-new new old ancient archaic old-fashioned",
 4: "spherical square rectangular circular triangular prismlike pyramidal",
 5: "purple pink red brown white yellow grey gray tan green blue black orange",
 6: "bangladeshi japanese thai egyptian chinese american indian german russian mexican brazilian turkish
 ↪  ethiopian nigerian filipino vietnamese iranian congolese pakistani indonesian",
 7: "lead steel silver cloth fiberglass wood cardboard rubber iron plastic glass paper leather wool gold",
 8: "exercise smoking snorkeling eating drinking hiking walking",
}
WORDS = {w: c for c, s in CATS.items() for w in s.split()}

def cat(w):
    lw = w.lower()
    if lw in WORDS:
        return WORDS[lw]
    if lw.endswith("ing"):
        return 8
    if w[:1].isupper():
        return 6
    return 1

def ok(phrase):
    words = phrase.split()[:-1]
    cs = [cat(w) for w in words]
    return all(a <= b for a, b in zip(cs, cs[1:]))

def solve(x):
    opts = re.findall(r"\(([A-Z])\)\s*([^\n]+)", x)
    for letter, phrase in opts:
        if ok(phrase.strip()):
            return "(" + letter + ")"
    return "(" + opts[0][0] + ")" if opts else "(A)"
```

Object counting. Llama3.1-70B

```
def solve(x):
    # Split the input string into individual items
    items = x.split(', ')

    # Initialize a counter for the total number of items
    total = 0

    # Iterate over each item
    for item in items:
        # Check if the item has a quantity specified
        if 'three' in item:
            total += 3
        elif 'four' in item:
            total += 4
        elif 'five' in item:
            total += 5
        elif 'two' in item:
            total += 2
```

```
        else:
            # If no quantity is specified, assume it's 1
            total += 1

    return total
```

Claude Opus 5.5

```
import re
NUM={'a':1,'an':1,'one':1,'two':2,'three':3,'four':4,'five':5,'six':6,'seven':7,'eight':8,'nine':9,'ten':10}
CATS={
'vegetable':['lettuce','cauliflower','yam','cabbage','broccoli','potato','potatoe','garlic','carrot','celery','onion'],
'fruit':['peach','raspberr','nectarine','plum','banana','strawberr','blackberr','orange','grape','apple'],
'animal':['fish','snake','mouse','mice','bear','rabbit','pig','goat','cow','snail','donkey','chicken','duck','cat','dog','frog'],
'musical instrument':['trombone','clarinet','drum','violin','piano','flute','accordion','trumpet'],
'object':['fridge','chair','microwave','stove','bed','car','couch','toaster','lamp','table','oven'],
}
def solve(x):
    m=re.search(r'I have (.*)\. How many (.+?) do I have',x)
    items,cat=m.group(1),m.group(2).lower()
    key=None
    for k in CATS:
        if cat.startswith(k): key=k
    total=0
    for part in re.split(r',\s*(?:and\s+)?|\s+and\s+',items):
        w=part.strip().split()
        if not w: continue
        n=NUM.get(w[0].lower(),1)
        noun=' '.join(w[1:]).lower()
        if key and any(re.search(r'\b'+s,noun) for s in CATS[key]):
            total+=n
    return str(total)
```

Formal fallacies. Llama3.1-70B

```
def solve(x):
    premises = x.split('. ')
    conclusion = premises[-1]
    premises = premises[:-1]

    conclusion = conclusion.replace('We may conclude that ', '').replace('We may conclude: ',
    ↪  '').replace('It follows that ', '').replace('So, necessarily, ', '').replace('All this entails
    ↪  that ', '').replace('In consequence, ', '')

    conclusion = conclusion.replace('necessarily ', '').replace('not ', 'not_').replace(' is ',
    ↪  '_is_').replace(' or ', '_or_').replace(' and ', '_and_').replace(' every ', 'every_').replace('
    ↪  whoever ', 'whoever_').replace(' being a ', 'being_a_').replace(' an ', 'an_').replace(' a ',
    ↪  'a_')

    premises = [premise.replace('First premise: ', '').replace('Second premise: ', '').replace('Third
    ↪  premise: ', '') for premise in premises]
    premises = [premise.replace('To start with, ', '').replace('To begin with, ', '').replace('First, ',
    ↪  '').replace('Moreover, ', '').replace('Next, ', '') for premise in premises]
    premises = [premise.replace('Now, ', '').replace('Now ', '') for premise in premises]

    premises = [premise.replace(' is ', '_is_').replace(' or ', '_or_').replace(' and ',
    ↪  '_and_').replace(' every ', 'every_').replace(' whoever ', 'whoever_').replace(' being a ',
    ↪  'being_a_').replace(' an ', 'an_').replace(' a ', 'a_') for premise in premises]
    premises = [premise.replace('not ', 'not_') for premise in premises]

    conclusion = conclusion.split(' ')
    premises = [premise.split(' ') for premise in premises]

    def is_valid(premises, conclusion):
        # Create a dictionary to store the premises
        premises_dict = {}
        for premise in premises:
            if 'not_' in premise[0]:
                premises_dict[premise[1]] = False
            elif premise[0] == 'every':
                premises_dict[premise[2]] = True
            elif premise[0] == 'whoever':
                premises_dict[premise[2]] = True
            elif '_is_' in premise:
                if premise[0] == 'not':
                    premises_dict[premise[2]] = False
                else:
                    premises_dict[premise[0]] = True

        # Check the conclusion
        if 'not_' in conclusion[0]:
```

```
            return premises_dict.get(conclusion[1], True) == False
        elif conclusion[0] == 'every':
            return premises_dict.get(conclusion[2], False) == True
        elif conclusion[0] == 'whoever':
            return premises_dict.get(conclusion[2], False) == True
        elif '_is_' in conclusion:
            if conclusion[0] == 'not':
                return premises_dict.get(conclusion[2], True) == False
            else:
                return premises_dict.get(conclusion[0], False) == True

    def is_valid_2(premises, conclusion):
        # Create a dictionary to store the premises
        premises_dict = {}
        for premise in premises:
            if '_or_' in premise:
                p1, p2 = premise
                p1 = p1.replace('not_', '')
                p2 = p2.replace('not_', '')
                premises_dict[p1] = True
                premises_dict[p2] = True
            elif '_and_' in premise:
                p1, p2 = premise
                p1 = p1.replace('not_', '')
                p2 = p2.replace('not_', '')
                premises_dict[p1] = True
                premises_dict[p2] = True
            elif 'not_' in premise[0]:
                premises_dict[premise[1]] = False
            elif premise[0] == 'every':
                premises_dict[premise[2]] = True
            elif premise[0] == 'whoever':
                premises_dict[premise[2]] = True
            elif '_is_' in premise:
                if premise[0] == 'not':
                    premises_dict[premise[2]] = False
                else:
                    premises_dict[premise[0]] = True

        # Check the conclusion
        if 'not_' in conclusion[0]:
            return premises_dict.get(conclusion[1], True) == False
        elif conclusion[0] == 'every':
            return premises_dict.get(conclusion[2], False) == True
        elif conclusion[0] == 'whoever':
            return premises_dict.get(conclusion[2], False) == True
        elif '_is_' in conclusion:
            if conclusion[0] == 'not':
                return premises_dict.get(conclusion[2], True) == False
            else:
                return premises_dict.get(conclusion[0], False) == True

    def is_valid_3(premises, conclusion):
        # Create a dictionary to store the premises
        premises_dict = {}
        for premise in premises:
            if '_or_' in premise:
                p1, p2 = premise
                p1 = p1.replace('not_', '')
                p2 = p2.replace('not_', '')
                premises_dict[p1] = True
                premises_dict[p2] = True
            elif '_and_' in premise:
                p1, p2 = premise
                p1 = p1.replace('not_', '')
                p2 = p2.replace('not_', '')
                premises_dict[p1] = True
                premises_dict[p2] = True
            elif 'not_' in premise[0]:
                premises_dict[premise[1]] = False
            elif premise[0] == 'every':
                premises_dict[premise[2]] = True
            elif premise[0] == 'whoever':
                premises_dict[premise[2]] = True
            elif '_is_' in premise:
                if premise[0] == 'not':
                    premises_dict[premise[2]] = False
                else:
                    premises_dict[premise[0]] = True

        # Check the conclusion
        if 'not_' in conclusion[0]:
            return premises_dict.get(conclusion[1], True) == False
```

```
        elif conclusion[0] == 'every':
            return premises_dict.get(conclusion[2], False) == True
        elif conclusion[0] == 'whoever':
            return premises_dict.get(conclusion[2], False) == True
        elif '_is_' in conclusion:
            if conclusion[0] == 'not':
                return premises_dict.get(conclusion[2], True) == False
            else:
                return premises_dict.get(conclusion[0], False) == True

    if is_valid(premises, conclusion) or is_valid_2(premises, conclusion) or is_valid_3(premises,
    ↪  conclusion):
        return 'valid'
    else:
        return 'invalid'
```

Claude Opus 5.5

```
import re, itertools

ART = r'(?:an? |not |neither |both |no )'

def _split(s, sep):
    return [p for p in re.split(sep, s)]

def parse_pred(s):
    s = s.strip().strip('.').strip()
    s = re.sub(r',\s*too$', '', s)
    s = re.sub(r'^(?:also|however) ', '', s)
    s = s.replace('or, otherwise,', 'or').replace('and, in the same time,', 'and')
    s = s.strip()
    parts = re.split(r',? or (?=' + ART + ')', s)
    if len(parts) > 1:
        return ('or', [parse_pred(p) for p in parts])
    if s.startswith('neither '):
        a, b = s[8:].split(' nor ', 1)
        return ('and', [('not', parse_pred(a)), ('not', parse_pred(b))])
    if s.startswith('not both '):
        return ('not', parse_pred('both ' + s[9:]))
    if s.startswith('both '):
        s = s[5:]
    parts = re.split(r',? and (?=' + ART + ')|, (?=' + ART + ')', s)
    if len(parts) > 1:
        return ('and', [parse_pred(p) for p in parts])
    if s.startswith('not '):
        return ('not', parse_pred(s[4:]))
    s = re.sub(r'^(?:an?|the) ', '', s)
    return ('atom', s.lower().strip())

def atoms(f, acc):
    if f[0] == 'atom': acc.add(f[1])
    elif f[0] == 'not': atoms(f[1], acc)
    elif f[0] in ('and', 'or'):
        for g in f[1]: atoms(g, acc)
    elif f[0] == 'imp':
        atoms(f[1], acc); atoms(f[2], acc)

def ev(f, t):
    k = f[0]
    if k == 'atom': return t[f[1]]
    if k == 'not': return not ev(f[1], t)
    if k == 'and': return all(ev(g, t) for g in f[1])
    if k == 'or': return any(ev(g, t) for g in f[1])
    if k == 'imp': return (not ev(f[1], t)) or ev(f[2], t)

def imp(a, b): return ('imp', a, b)
def neg(a): return ('not', a)

QW = r'(?:everyone|everybody|everything|whoever|whatever|anyone|anything|anybody)'

def parse_sent(s):
    s = s.strip().strip('.').strip()
    s = re.sub(r'^(?:necessarily|then|that),?\s+', '', s)
    s = s.replace('or, otherwise,', 'or').replace('and, in the same time,', 'and')
    if s and s[0].isupper() and not re.match(r'^[A-Z][a-z]+ is ', s) or True:
        pass
    low = s[0].lower() + s[1:] if s else s
    m = re.match(r'^it is (?:false|not the case) that (.+?) is (.+)$', low)
    if m:
        return ('ind', m.group(1).strip(), neg(parse_pred(m.group(2))))
    m = re.match(r'^nobody is neither (.+?) nor (.+)$', low)
    if m:
        return ('all', ('or', [parse_pred(m.group(1)), parse_pred(m.group(2))]))
```

```
    m = re.match(r'^(?:nobody|no one|nothing) is (.+)$', low)
    if m:
        return ('all', neg(parse_pred(m.group(1))))
    m = re.match(r'^no (.+?) is (.+)$', low)
    if m:
        subj = m.group(1)
        subs = re.split(r' and no ', subj)
        sf = [parse_subj(x) for x in subs]
        S = sf[0] if len(sf) == 1 else ('or', sf)
        return ('all', imp(S, neg(parse_pred(m.group(2)))))
    m = re.match(r'^(?:every|each|any) (.+?) (?:that|who) is (.+?) is (?:also )?(.+)$', low)
    if m:
        return ('all', imp(('and', [parse_pred(m.group(1)), parse_pred(m.group(2))]),
        ↪  parse_pred(m.group(3))))
    m = re.match(r'^(?:every|each|any) (.+?) is (.+)$', low)
    if m:
        return ('all', imp(parse_pred(m.group(1)), parse_pred(m.group(2))))
    m = re.match(r'^' + QW + r' is none of this: (.+), is (.+)$', low)
    if m:
        lst = m.group(1)
        i = lst.rfind(' or ')
        a, b = lst[:i], lst[i + 4:]
        return ('all', imp(('and', [neg(parse_pred(a)), neg(parse_pred(b))]), parse_pred(m.group(2))))
    m = re.match(r'^' + QW + r' (?:who |that )?is (.+?),? is (?:also |however )?(.+)$', low)
    if m:
        return ('all', imp(parse_pred(m.group(1)), parse_pred(m.group(2))))
    m = re.match(r'^(not )?being (.+?) is (sufficient|necessary) for (not )?being (.+)$', low)
    if m:
        a = parse_pred(m.group(2)); b = parse_pred(m.group(5))
        if m.group(1): a = neg(a)
        if m.group(4): b = neg(b)
        if m.group(3) == 'sufficient':
            return ('all', imp(a, b))
        return ('all', imp(b, a))
    m = re.match(r'^if (?:someone|somebody|something|anyone) is (.+?),? then (?:he|she|that person|they|it)
    ↪  (?:is|are) (.+)$', low)
    if m:
        return ('all', imp(parse_pred(m.group(1)), parse_pred(m.group(2))))
    m = re.match(r'^(?:there is|there exists|there are) (?:somebody|someone|something|anybody) (?:who|that)
    ↪  is (.+)$', low)
    if m:
        return ('some', parse_pred(m.group(1)))
    m = re.match(r'^(?:somebody|someone|something) is (.+)$', low)
    if m:
        return ('some', parse_pred(m.group(1)))
    m = re.match(r'^(?:there is|there exists) (?:an? )?(.+?) (?:who|that) is (.+)$', low)
    if m:
        return ('some', ('and', [parse_pred(m.group(1)), parse_pred(m.group(2))]))
    m = re.match(r'^some (.+?) (?:who|that) is (.+?) is (.+)$', low)
    if m:
        return ('some', ('and', [parse_pred(m.group(1)), parse_pred(m.group(2)), parse_pred(m.group(3))]))
    m = re.match(r'^some (.+?) is (.+)$', low)
    if m:
        return ('some', ('and', [parse_pred(m.group(1)), parse_pred(m.group(2))]))
    m = re.match(r'^(.+?) is (.+)$', s)
    if m:
        return ('ind', m.group(1).strip(), parse_pred(m.group(2)))
    return None

def parse_subj(x):
    return parse_pred(x)

CONC = [r'it follows that', r'we may conclude that', r'we may conclude:', r'we may conclude',
        r'therefore,', r'in consequence,', r'all this entails that', r'so, necessarily,',
        r'hence,', r'from this follows:', r'consequently,', r'thus,', r'it is therefore',
        r'which derives from the fact that', r'so,']
PREM = [r'to start with,', r'to begin with,', r'first of all,', r'first premise:', r'second premise:',
        r'third premise:', r'fourth premise:', r'first,', r'second,', r'third,', r'next,', r'plus,',
        r'now,', r'moreover,', r'finally,', r'furthermore,', r'additionally,', r'besides,',
        r'in addition,', r'also,', r'what is more,', r'all in all,', r'lastly,']

def solve(x):
    try:
        return _solve(x)
    except Exception:
        return 'valid'

def _solve(x):
    t = x
    i = t.find('argument')
    j = t.find(':', i)
    body = t[j + 1:]
    k = body.rfind('"')
```

```
    if k > 0: body = body[:k]
    body = body.strip()
    lb = body.lower()
    best = None
    for c in CONC:
        for mm in re.finditer(re.escape(c), lb):
            if best is None or mm.start() < best[0] or (mm.start() == best[0] and mm.end() > best[1]):
                best = (mm.start(), mm.end())
            break
    prem_txt = body[:best[0]]; conc_txt = body[best[1]:]
    prem_txt = prem_txt.replace('St. ', 'St# ')
    sents = [p for p in re.split(r'\.\s*', prem_txt) if p.strip()]
    prems = []
    for p in sents:
        p = p.strip().replace('St# ', 'St. ')
        changed = True
        while changed:
            changed = False
            for pm in PREM:
                if p.lower().startswith(pm):
                    p = p[len(pm):].strip(); changed = True
        prems.append(parse_sent(p))
    conc = parse_sent(conc_txt.strip())
    allf = [p for p in prems + [conc] if p]
    A = set()
    for p in allf: atoms(p[-1], A)
    A = sorted(A)
    types = [dict(zip(A, v)) for v in itertools.product([False, True], repeat=len(A))]
    T = [ty for ty in types if all(ev(p[1], ty) for p in prems if p and p[0] == 'all')]
    if conc[0] == 'some':
        T = [ty for ty in T if not ev(conc[1], ty)]
    if not T: return 'valid'
    consts = {}
    for p in prems:
        if p and p[0] == 'ind': consts.setdefault(p[1], []).append(p[2])
    if conc[0] == 'ind': consts.setdefault(conc[1], []).append(neg(conc[2]))
    for p in prems:
        if p and p[0] == 'some':
            if not any(ev(p[1], ty) for ty in T): return 'valid'
    for c, fs in consts.items():
        if not any(all(ev(f, ty) for f in fs) for ty in T): return 'valid'
    if conc[0] == 'all':
        if not any(not ev(conc[1], ty) for ty in T): return 'valid'
    return 'invalid'
```

Disambiguation QA. Llama3.1-70B

```
def solve(sentence, options):
    # Tokenize the sentence into words
    words = sentence.split()

    # Find the pronoun in the sentence
    pronouns = ['he','she', 'they', 'him', 'her', 'them']
    pronoun = None
    for word in words:
        if word.lower() in pronouns:
            pronoun = word
            break

    # If no pronoun is found, return Ambiguous
    if pronoun is None:
        return 'C'

    # Determine the antecedent of the pronoun
    if pronoun.lower() == 'they':
        # 'they' can refer to a single person or a group, so it's often ambiguous
        if 'and' in sentence:
            return 'C'
        else:
            # If 'they' is used with a single person, it's likely referring to that person
            for i, word in enumerate(words):
                if word.lower() == 'they':
                    if i > 0 and words[i-1].lower() in ['the', 'a', 'an']:
                        return 'A'
                    else:
                        return 'B'
    elif pronoun.lower() in ['he', 'him']:
        # 'he' and 'him' usually refer to a male
        for i, word in enumerate(words):
            if word.lower() == 'he' or word.lower() == 'him':
                if i > 0 and words[i-1].lower() in ['the', 'a', 'an']:
                    return 'A'
```

```
                else:
                    return 'B'
    elif pronoun.lower() in ['she', 'her']:
        #'she' and 'her' usually refer to a female
        for i, word in enumerate(words):
            if word.lower() =='she' or word.lower() == 'her':
                if i > 0 and words[i-1].lower() in ['the', 'a', 'an']:
                    return 'A'
                else:
                    return 'B'

    # If the pronoun is not found in the sentence, return Ambiguous
    return 'C'
```

Claude Opus 5.5

```
import re

RULES = [
    ('office', 'C'), ('discuss their favorite', 'C'), ('discuss his favorite', 'C'), ('discuss her
    ↪  favorite', 'C'),
    ('shared a story', 'C'), ('share a story', 'C'), ('shares a story', 'C'),
    ('had cancer', 'C'), ('needed more time', 'C'), ('standing in front', 'C'),
    ('sent the letter', 'C'), ('used big words', 'C'), ('use big words', 'C'), ('arrogant', 'C'),
    ('unable to get in contact', 'C'),
    ('future payments', 'B'), ('could help with tax', 'B'), ('gravely ill', 'B'), ('felt ill', 'B'),
    ('sick', 'B'), ('culinary training', 'B'), ('blueprints', 'B'), ('needed one', 'B'),
    ('may be guilty', 'B'), ('was guilty', 'B'), ('got fined', 'B'), ('get fined', 'B'),
    ('reply', 'B'), ('idea', 'B'), ('needed a lab assistant', 'B'), ('need a lab assistant', 'B'),
    ('stellar performance', 'B'), ('wrote a good book', 'B'), ('smelled awful', 'B'),
    ('might fall', 'B'), ('should floss', 'B'), ('needed to get more rest', 'B'),
    ('should avoid', 'B'), ('send the prices', 'B'), ('need any help', 'B'), ('needed any help', 'B'),
    ('could help', 'B'), ('can help', 'B'),
    ('good mood', 'A'), ('better understood', 'A'), ('too late', 'A'),
    ('working on', 'A'), ('told us that', 'A'), ('tells us that', 'A'),
    ('took the day off', 'A'), ('skin condition', 'A'), ('purchased', 'A'),
    ('forgot to bring', 'A'), ('very busy', 'A'), ('grading policy', 'A'), ('like the cake', 'A'),
    ('likes the cake', 'A'), ('understand the case', 'A'), ('borrow some money', 'A'),
    ('were fired', 'A'), ('was fired', 'A'), ('got fired', 'A'), ('upheld the peace', 'A'),
    ('more experience', 'A'), ('microscope', 'A'), ('filing his taxes', 'A'), ('filing her taxes', 'A'),
    ('filing their taxes', 'A'), ('like to teach', 'A'), ('likes to teach', 'A'),
    ('did not like the design', 'A'), ("didn't like the design", 'A'), ('repairing', 'A'),
    ('history of', 'A'), ('focus on code', 'A'), ('focuses on code', 'A'), ('anything they could help',
    ↪  'A'),
    ('their secretary', 'A'), ('his secretary', 'A'), ('her secretary', 'A'),
]

def solve(x):
    m = re.search(r'Sentence:\s*(.*?)\n', x)
    s = (m.group(1) if m else x).lower()
    for k, v in RULES:
        if re.search(r'\b' + re.escape(k) + r'\b', s):
            return '(%s)' % v
    if re.search(r'\b(their|his|her)\b', s) and not re.search(r'\b(him|them)\b', s):
        return '(C)'
    if re.search(r'\b(him|her|them)\b', s):
        return '(B)'
    if re.search(r', (and|but) (he|she|they)\b', s):
        return '(C)'
    return '(A)'
```

Snarks. Qwen2.5-Coder-32B

```
def solve(x):
    # Split the input into lines to separate the question and options
    lines = x.strip().split('\n')
    # The question is the first line
    question = lines[0]
    # The options start from the third line and go until the end
    options = lines[2:]
    # Initialize a variable to keep track of the sarcastic option
    sarcastic_option = None

    # Iterate over the options to find the sarcastic one
    for option in options:
        # Split the option into label and statement
        label, statement = option.split(') ', 1)
        # Check for sarcastic indicators in the statement
```

```
        if any(indicator in statement for indicator in ['ARE U INSANE', 'Have my down vote', 'naive',
        ↪  'terrible support', 'pragmatic', 'more expensive', 'less fun', 'liability', 'enjoyable',
        ↪  'best', 'reddit', 'fun', 'sexist', 'win you points', 'lose you points', 'True', 'rational',
        ↪  'threaten', 'pleaded', 'highest', 'lowest', 'rich', 'loans', 'suspicious', 'legit',
        ↪  'perfect']):
            # If the statement contains a sarcastic indicator, mark it as the sarcastic option
            sarcastic_option = label
            break

    # Return the label of the sarcastic option in the required format
    return f"<FINAL_ANSWER>\n{label}\n</FINAL_ANSWER>"
```

Claude Opus 5.5

```
import re, difflib

SARC = ["megaconglomerate", "no", "years", "down vote", "person", "worthless", "asus's",
        "more", "generosity", "ever", "usually", "going up", "pleased", "never seen",
        "concrete", "devastating", "10000", "essential", "you don't", "?",
        "obviously", "great", "love", "totally", "sure", "always", "everyone", "best",
        "brilliant", "genius", "perfect", "wonderful", "fun"]
SINCERE = ["exorbitant", "less", "rarely", "displeased", "insufficient", "detrimental",
           "valuable", "liability", "several", "days", "up vote", "startup", "my",
           "self-centeredness", "going down", "i've seen", "stained-glass", "2"]

def _opts(x):
    m = re.search(r"\(A\)\s*(.*?)\s*\(B\)\s*(.*)", x, re.S)
    if not m:
        return "", ""
    return m.group(1).strip(), m.group(2).strip()

def _tok(s):
    return re.findall(r"[\w'\-]+|[^\w\s]", s.lower())

def _diff_text(a, b):
    ta, tb = _tok(a), _tok(b)
    sm = difflib.SequenceMatcher(None, ta, tb, autojunk=False)
    da, db = [], []
    for op, i1, i2, j1, j2 in sm.get_opcodes():
        if op == "equal":
            continue
        da.append(" ".join(ta[max(0, i1-1):i2+1]))
        db.append(" ".join(tb[max(0, j1-1):j2+1]))
    return " | ".join(da), " | ".join(db), i2 - i1 if False else None

def _score(d, core):
    s = 0
    words = " " + d + " "
    for k in SARC:
        if re.search(r"(?<![\w'\-])" + re.escape(k) + r"(?![\w'\-])", d):
            s += 2 if (" " in k or k in core) else 1
    for k in SINCERE:
        if re.search(r"(?<![\w'\-])" + re.escape(k) + r"(?![\w'\-])", d):
            s -= 2 if (" " in k or k in core) else 1
    return s

def solve(x):
    try:
        a, b = _opts(x)
        ta, tb = _tok(a), _tok(b)
        sm = difflib.SequenceMatcher(None, ta, tb, autojunk=False)
        da, db, ca, cb = [], [], [], []
        for op, i1, i2, j1, j2 in sm.get_opcodes():
            if op == "equal":
                continue
            da.append(" ".join(ta[max(0, i1-1):i2+1]))
            db.append(" ".join(tb[max(0, j1-1):j2+1]))
            ca += ta[i1:i2]; cb += tb[j1:j2]
        sa = _score(" | ".join(da), ca)
        sb = _score(" | ".join(db), cb)
        if sa > sb:
            return "(A)"
        if sb > sa:
            return "(B)"
        # fallback: shorter (fewer qualifiers) option tends to be sarcastic
        return "(A)" if len(a) <= len(b) else "(B)"
    except Exception:
        return "(A)"
```

Translation error. Llama3.1-70B

```
def solve(source, translation):
```

```
    # Check for Named Entities error
    if any(word in translation.lower() and word not in source.lower() for word in ['russia', 'new-york',
    ↪  'philly', 'eduard', 'kumbh', 'rockets', 'germany']):
        return 'D'

    # Check for Numerical Values error
    if any(char.isdigit() and source.count(char)!= translation.count(char) for char in source):
        return 'B'

    # Check for Modifiers or Adjectives error
    if any(word in translation.lower() and word not in source.lower() for word in ['non',
    ↪  'commercial','sea', 'gold', 'video']):
        return 'A'

    # Check for Negation or Antonyms error
    if 'no' in translation.lower() and 'allen' in source.lower():
        return 'C'

    # Check for Facts error
    if'sea' in translation.lower() and 'fluss' in source.lower():
        return 'F'

    # Check for Dropped Content error
    if len(source) - len(translation) > 20:
        return 'E'

    # If none of the above conditions are met, return 'D' as default
    return 'D'
```

Claude Opus 5.5

```
import re, difflib

def solve(x):
    try:
        src = re.search(r'Source:\s*(.*?)\nTranslation:', x, re.S).group(1).strip()
        tr = re.search(r'Translation:\s*(.*?)\nThe translation contains', x, re.S).group(1).strip()
    except Exception:
        return "(E)"
    sl, tl = src.lower(), tr.lower()
    # Negation / antonyms
    neg_t = re.search(r"\b(not|no|never|n't)\b", tl)
    neg_s = re.search(r"\b(nicht|kein\w*|nie)\b", sl)
    if bool(neg_t) != bool(neg_s):
        return "(C)"
    for w in ['hating', 'wide', 'smallest', 'least', 'worst', 'lowest', 'oldest', 'youngest']:
        if re.search(r'\b' + w + r'\b', tl):
            return "(C)"
    # Numbers
    sn = set(re.findall(r'\d+', src)); tn = set(re.findall(r'\d+', tr))
    if tn - sn:
        return "(B)"
    if sn - tn and not re.search(r'\b(\d+)(st|nd|rd|th)\b', tr):
        return "(E)"
    ords = {'erst': 'first', 'zweit': 'second', 'dritt': 'third', 'viert': 'fourth'}
    for g, e in ords.items():
        if re.search(r'\b' + g, sl) and not re.search(r'\b' + e + r'\b', tl):
            if any(re.search(r'\b' + e2 + r'\b', tl) for e2 in ords.values()) or 'erste bedeutende' in sl:
                return "(B)"
    # Single-letter change
    sl1 = set(re.findall(r'\b([A-Z])\b', src)); tl1 = set(re.findall(r'\b([A-Z])\b', tr))
    if tl1 - sl1:
        return "(F)"
    # Named entities
    stoks = [s for s in re.findall(r'[\w\-]+', src) if s[0].isupper() and len(s) >= 4]
    ttoks = re.findall(r'[\w\-]+', tr)
    for i, t in enumerate(ttoks):
        if not t[0].isupper() or len(t) < 4:
            continue
        tlw = t.lower()
        if tlw in sl:
            continue
        if any(s.lower() in tlw for s in stoks):
            continue
        best = max((difflib.SequenceMatcher(None, tlw, s.lower()).ratio() for s in stoks), default=0)
        if best >= 0.5:
            return "(D)"
    if 'new-york' in tl:
        return "(D)"
    # Parenthetical dropped
    if '(' in src and '(' not in tr:
        return "(A)"
    # Dropped content
```

```
if tr[0].islower():
    return "(E)"
ratio = len(tr.split()) / max(1, len(src.split()))
if ratio < 0.9:
    return "(E)"
return "(F)"
```

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: We provides formalization in Sections 3 and 4 and empirical verifications in Section 5.

   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: We discuss limitations in Section 6.

   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: [Yes]

Justification: We provide assumptions for each formalization in Sections 3 and 4.

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: We provide all experiment details in Sections B and 5.

   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: [Yes]

Justification: The link to the code repository is provided in the abstract.

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: We provide the full experimental setting and details in Sections B and 5.

   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: We report confidence intervals and standard deviation across random seeds and data samples in Figures 2 to 4.

   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: We provide information about computer resources in Section B.

   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: Our work does not involve human participants, sensitive data, or deployment in real-world systems, and therefore does not raise ethical concerns under the NeurIPS Code of Ethics.

   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: [Yes]

    Justification: This work proposes code as a compact and verifiable alternative to free-text explanations and aims to produce more faithful and verifiable explanations for LLMs. We do not anticipate direct negative impacts, as the work focuses on improving interpretability.

    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: We do not release any data or models.

    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: [Yes]

    Justification: We use open-source models and public datasets, Claude Opus 5.5 accessed through the Anthropic API, and cite the sources for all models and datasets in Section 5.

    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: [Yes]

    Justification: The link to the code repository is provided in the abstract.

    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: Our paper does not involve crowdsourcing nor research with human subjects.

    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: Our paper does not involve crowdsourcing nor research with human subjects.

    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: [N/A]

Justification: The core method development in this research does not involve LLMs as any important, original, or non-standard components.

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.