On the Stability of Code Generation under Semantically Equivalent Intermediate Explanations

Mingyang Xie
MIB Laboratory, Queen's University
21mx4@queensu.ca

Xiangyu Jiao
MIB Laboratory, Queen's University
20xj6@queensu.ca

Chuyi Qin
MIB Laboratory, Queen's University
21ciq@queensu.ca

Ting Hu*
MIB Laboratory, Queen's University
ting.hu@queensu.ca

Abstract

Large language models (LLMs) achieve strong performance in code generation, yet the stability of their behavior in multi stage coding pipelines remains insufficiently understood. In particular, it is unclear whether semantically equivalent, model generated explanations, often treated as reliable intermediate representations, lead to consistent executable code. We present an evaluation framework that factorizes the pipeline into Problem to Explanation to Code to Execution, and quantify stability at both the explanation and execution levels. We curate a benchmark of algorithmic problems spanning multiple difficulty tiers and classical machine learning algorithms, selected to balance practical relevance and reduced memorization likelihood. For each problem, the model generates k semantically equivalent explanations in isolated interactions, each used in a separate interaction to generate code. We evaluate functional correctness with an online judge style test harness, and introduce a structure aware, sequence sensitive semantic consistency metric designed for code reconstructability rather than surface overlap. Our results show that small variations in intermediate explanations can be amplified into large differences in code correctness, revealing a distinct and practically important instability mode beyond user prompt reformulations.

1 Introduction

Large language models (LLMs) have demonstrated strong performance in code generation and are increasingly used in practical development workflows. Most evaluations measure functional correctness under a fixed problem description, typically via unit tests or online judge style scoring and metrics such as pass at k [Liu et al., 2023]. These benchmarks have been valuable for tracking progress, but they largely treat code generation as a single step mapping from a problem statement to executable code, and therefore provide limited insight into the reliability of multi stage systems built around LLMs.

In practical LLM assisted programming workflows, code generation is often embedded in pipelines that explicitly produce and consume intermediate artifacts. A common pattern is to first generate an explanation, plan, or algorithmic sketch and then use it to drive code synthesis, refinement, or debugging [Lei et al., 2025]. These intermediate artifacts are frequently used as informal specifications: they are expected to preserve task semantics, highlight key algorithmic decisions, and constrain the space of correct implementations. Such pipelines are appealing because they appear to improve interpretability and controllability. However, they also introduce an additional and poorly understood

*Corresponding author

failure mode. Small semantic deviations in an intermediate explanation can be amplified downstream into incorrect code, even when the explanation appears reasonable or superficially equivalent. Understanding when and why this amplification happens is critical for deploying LLM based coding assistants in settings where correctness and reliability matter.

Recent work on prompt sensitivity and robustness has shown that small perturbations to user facing prompts can lead to substantial variation in LLM outputs and performance [Cao et al., 2024]. While informative, this line of work primarily studies external reformulations of the input prompt. In contrast, our focus is on model generated intermediate representations that arise inside the pipeline and are implicitly assumed to be canonical and interchangeable. The question is not whether different user phrasings change model behavior, but whether semantically equivalent intermediate explanations generated by the model itself lead to stable executable outcomes when reused as inputs to code synthesis. This distinction matters in practice because intermediate artifacts are increasingly reused, shared, and cached in LLM based tools, and their reliability directly determines pipeline robustness.

This paper studies explanation level stability in LLM based code generation through a two step setting: for a given coding problem, the model generates k semantically equivalent explanations in isolated interactions without shared context, and each explanation is then used in a separate interaction to generate executable code. The core methodological challenge is that semantic equivalence in this setting cannot be captured by surface level textual overlap. In code generation, certain components of an explanation, such as invariants, boundary conditions, and the ordering of algorithmic steps, are disproportionately important for reconstructing a correct program. Therefore, analyzing stability at the explanation level requires a task aligned measure of explanation consistency that is sensitive to structure and sequence rather than purely lexical similarity.

To enable systematic analysis, we present an evaluation framework that factorizes the pipeline into Problem to Explanation to Code to Execution, and quantify stability at both the explanation and execution levels. We curate a benchmark of algorithmic problems spanning multiple difficulty tiers of competitive programming as well as classical machine learning algorithms. Problems are selected to balance practical relevance with reduced likelihood of memorization, so that observed behaviors more plausibly reflect reasoning and generalization rather than recall. For each problem, we evaluate functional correctness using an online judge style harness with predefined test cases. In parallel, we introduce a structure aware, sequence sensitive semantic consistency metric designed as a task aligned proxy for code reconstructability, and aggregate pairwise similarities within each explanation set to characterize explanation stability.

Our analysis reveals that even when explanations are highly consistent under the proposed semantic metric, the resulting code can vary substantially in correctness. This suggests that seemingly minor semantic shifts in intermediate explanations can be amplified into execution failures, exposing a distinct and practically important instability mode that is not captured by standard prompt sensitivity evaluations. We further observe systematic differences in stability across task difficulty levels and model configurations, motivating explanation level stability as a complementary dimension for evaluating and deploying code generating language models.

2 Problem Setup and Formalization

We study explanation level stability in a two stage LLM based coding pipeline. Let P denote a programming problem, consisting of a natural language specification and optional examples. For each problem P, we generate a set of k intermediate explanations $E(P) = \{e_1, \ldots, e_k\}$, where each e_i is intended to be semantically equivalent and sufficient to reconstruct a correct solution. Each explanation e_i is produced in an isolated interaction with no shared conversational context. In a second isolated interaction, the model takes e_i as input and produces a program $c_i = G(e_i)$, where $G(\cdot)$ denotes the code synthesis step. This defines the pipeline $P \to e_i \to c_i$.

To evaluate functional correctness, we use an online judge style test harness. For each problem P, we define a set of test cases $T(P) = \{(x_j, y_j)\}_{j=1}^{m}$, where x_j is an input and y_j is the expected output. Executing program c_i on x_j yields an output $\hat{y}_{ij}$. We define the per test case correctness indicator $r_{ij} = \mathbb{I}[\hat{y}_{ij} = y_j]$, and the code correctness score as

$$A(P, c_i) = \frac{1}{m} \sum_{j=1}^{m} r_{ij}, \tag{1}$$

which lies in $[0, 1]$ and can be interpreted as the fraction of passed test cases. The execution level stability of the pipeline for problem P is then characterized by the distribution $\{A(P, c_i)\}_{i=1}^{k}$, e.g., its mean, variance, or failure rate.

A central challenge is to quantify whether the explanation set $E(P)$ is semantically consistent in a way that is relevant to code reconstructability. Surface level overlap metrics are inadequate, and even general semantic similarity measures may fail to capture algorithmically critical content such as invariants, boundary conditions, and step ordering [Zhang et al., 2020]. We therefore define an explanation level consistency score that is structure aware and sequence sensitive. Concretely, we assume each explanation e can be mapped to a structured representation $\phi(e)$ with a fixed set of fields (e.g., key ideas, algorithm steps, constraints, and edge cases). Let $S(\phi(e_i), \phi(e_j)) \in [0, 1]$ denote a weighted similarity function over structured fields, optionally incorporating sequence alignment over the algorithm steps. We then define the explanation set consistency as the average pairwise similarity:

$$\mathrm{Cons}(P) = \frac{1}{k(k-1)} \sum_{i \neq j} S(\phi(e_i), \phi(e_j)). \tag{2}$$

Finally, our objective is to analyze the relationship between explanation consistency and execution outcomes. For each problem P, we jointly consider (i) $\mathrm{Cons}(P)$ as an estimate of intermediate representation stability, and (ii) the correctness distribution $\{A(P, c_i)\}_{i=1}^{k}$ as the execution level behavior of the pipeline. In later sections, we study when high explanation consistency fails to guarantee stable correctness, and quantify the extent to which small semantic deviations in intermediate explanations are amplified into execution failures.

3 Method

3.1 Overview

We evaluate explanation level stability in a two stage LLM based coding pipeline. Given a programming problem P, an LLM first generates a set of k semantically equivalent intermediate explanations $E(P) = \{e_1, \ldots, e_k\}$ in isolated interactions. Each explanation e_i is then used as the sole conditioning input in a separate, context isolated interaction to synthesize executable code $c_i = G(e_i)$. We treat e_i as a latent program specification: an informal intermediate representation expected to encode algorithmic decisions, invariants, and boundary conditions required for a correct implementation. Our methodology quantifies stability at both (i) the execution level via an online judge style test harness and (ii) the explanation level via a learned semantic similarity model tailored to structured algorithm descriptions.

3.2 Problem Set Construction

We construct a problem set of $n = 18$ tasks with four categories: competitive programming problems at three difficulty tiers (easy, medium, hard) and classical machine learning algorithms. Specifically, we select 5 problems per tier (15 total) from Luogu, LeetCode, and Codeforces, and 3 classic machine learning algorithm tasks. The selection criteria are: (1) the tasks are practically relevant and not overly niche, (2) the tasks are not extremely high frequency or trivially canonicalized in a way that makes memorization likely, and (3) each task admits a clear reference solution and can be evaluated via a finite set of test cases. For each task, we build a test suite that includes both typical and edge scenarios, following the standard online judge evaluation paradigm.

3.3 Generating Semantically Equivalent Explanations as Structured Specifications

For each problem P, we prompt the LLM to generate k semantically equivalent explanations. Each explanation is generated in a fresh interaction with no shared context and no conversational memory. The prompt requires the model to preserve core semantics while varying phrasing, and to output a structured explanation under a fixed schema. The schema is designed to reflect code reconstructability and includes fields such as: (a) key algorithmic idea, (b) stepwise algorithm procedure, (c) invariants and constraints, and (d) boundary conditions and edge cases. We denote the structured representation of an explanation by $\phi(e)$. This representation serves two purposes: it reduces ambiguity for downstream code synthesis and provides a stable interface for similarity estimation, since algorithmic components should dominate similarity judgments more than surface level wording.

3.4 Explanation Conditioned Code Synthesis

Given an explanation e_i, we synthesize code in a separate isolated interaction. The code generation prompt instructs the model to implement the algorithm specified by e_i, adhere to the problem input output format, and output a complete executable program under a fixed language and runtime setting. Each explanation is used independently, producing a set of programs $\{c_i\}_{i=1}^{k}$ for the same problem. We standardize decoding and formatting constraints to ensure compatibility with the evaluation harness, and log model identifiers and generation parameters for reproducibility.

3.5 Multi Model and Multi Temperature Experimental Design

To assess generality, we evaluate multiple frontier LLMs, including GPT-5.2, LLaMA, and Gemini. For each model, we repeat the entire pipeline across multiple temperature settings to quantify the effect of sampling stochasticity on both explanation stability and code correctness. This yields a matrix of conditions indexed by (model, temperature, problem). For each condition, we generate k explanations and k codes, enabling within condition measurement of variability and between condition comparison across models and temperatures.

3.6 Online Judge Style Functional Evaluation

We evaluate functional correctness using an online judge style harness. For each problem P, we define a set of test cases $T(P) = \{(x_j, y_j)\}_{j=1}^{m}$. Each synthesized program c_i is executed on all inputs x_j, and its output $\hat{y}_{ij}$ is compared against the expected output y_j. We define a binary indicator $r_{ij} = \mathbb{I}[\hat{y}_{ij} = y_j]$, and compute the code correctness score:

$$A(P, c_i) = \frac{1}{m} \sum_{j=1}^{m} r_{ij}. \tag{3}$$

Execution level stability for a problem is characterized by the distribution $\{A(P, c_i)\}_{i=1}^{k}$, and aggregated by category and experimental condition. In addition to correctness scores, we record structured failure types from the harness, such as wrong answer, runtime error, and time limit exceeded, to support error taxonomy analysis.

3.7 Learning a Task Aligned Explanation Similarity Model

To measure semantic consistency of structured explanations, we use a cross encoder initialized from `cross-encoder/stsb-roberta-large` and fine tune it for similarity estimation on algorithm description texts. The training objective is regression to a similarity score in $[0, 1]$, and the training data are constructed to reflect code reconstructability rather than generic semantic overlap. Concretely, we curate paired samples of structured algorithm descriptions with labels that prioritize agreement on algorithmic decisions, invariants, and edge cases, while de emphasizing superficial paraphrases and variable naming. We further include contrastive pairs where lexical overlap is high but the underlying algorithm differs, and pairs where the algorithm is the same but the step ordering or boundary conditions differ. This fine tuned cross encoder outputs a similarity score

$$s_{ij} = S(\phi(e_i), \phi(e_j)) \in [0, 1]. \tag{4}$$

3.8 Explanation Set Consistency and Joint Stability Analysis

For each problem P under a given (model, temperature) condition, we compute pairwise similarities for all distinct pairs (i, j), and define the explanation set consistency as the average pairwise similarity:

$$\mathrm{Cons}(P) = \frac{1}{k(k-1)} \sum_{i \neq j} s_{ij}. \tag{5}$$

We then analyze the relationship between explanation stability and execution outcomes by pairing $\mathrm{Cons}(P)$ with the correctness distribution $\{A(P, c_i)\}_{i=1}^{k}$. This joint view allows us to test whether intermediate representations that are stable under the semantic metric also yield stable code correctness, and to identify cases where small semantic drifts in explanations are amplified into execution failures.

3.9 Testable Hypotheses on Drift and Failure Amplification

We operationalize the latent specification view into three testable hypotheses, each mapped to a concrete analysis.

H1 (Information bearing differences). Differences among explanations primarily occur in algorithmic decisions, invariants, and edge cases rather than surface paraphrasing. We test H1 by decomposing the structured representation $\phi(e)$ into fields and measuring field level similarity distributions across explanation pairs. If H1 holds, variance should concentrate in fields corresponding to decisions, invariants, and boundary conditions.

H2 (Mediating role of explanation drift). Controlling for prompt templates, model identity, and temperature, explanation differences measured by $\mathrm{Cons}(P)$ or field level consistency significantly predict code pass or fail outcomes, and explain part of the effect from generation conditions to code correctness. We test H2 using regression and mediation style analyses where correctness indicators are modeled as a function of explanation consistency and generation conditions, and quantify whether explanation consistency accounts for a significant portion of outcome variability.

H3 (Systematic failure modes). Low explanation consistency is associated with structured error types such as missing boundary conditions, incorrect state transitions, or wrong loop direction, rather than random noise. We test H3 by mapping failing programs to an error taxonomy and comparing the distribution of error categories between high consistency and low consistency regimes. If H3 holds, low consistency cases should exhibit higher rates of systematic logical failures aligned with specification drift.

4 !! Not Real Content Below | General formatting instructions

The text must be confined within a rectangle 5.5 inches (33 picas) wide and 9 inches (54 picas) long. The left margin is 1.5 inch (9 picas). Use 10 point type with a vertical spacing (leading) of 11 points. Times New Roman is the preferred typeface throughout, and will be selected for you by default. Paragraphs are separated by ½ line space (5.5 points), with no indentation.

The paper title should be 17 point, initial caps/lower case, bold, centered between two horizontal rules. The top rule should be 4 points thick and the bottom rule should be 1 point thick. Allow ¼ inch space above and below the title to rules. All pages should start at 1 inch (6 picas) from the top of the page.

For the final version, authors' names are set in boldface, and each name is centered above the corresponding address. The lead author's name is to be listed first (left-most), and the co-authors' names (if different address) are set to follow. If there is only one co-author, list both author and co-author side by side.

Please pay special attention to the instructions in Section 6 regarding figures, tables, acknowledgments, and references.

5 Headings: first level

All headings should be lower case (except for first word and proper nouns), flush left, and bold.

First-level headings should be in 12-point type.

5.1 Headings: second level

Second-level headings should be in 10-point type.

5.1.1 Headings: third level

Third-level headings should be in 10-point type.

Paragraphs There is also a `\paragraph` command available, which sets the heading in bold, flush left, and inline with the text, with the heading followed by 1 em of space.

6 Citations, figures, tables, references

These instructions apply to everyone.

6.1 Citations within the text

The `natbib` package will be loaded for you by default. Citations may be author/year or numeric, as long as you maintain internal consistency. As to the format of the references themselves, any style is acceptable as long as it is used consistently.

The documentation for `natbib` may be found at

`http://mirrors.ctan.org/macros/latex/contrib/natbib/natnotes.pdf`

Of note is the command `\citet`, which produces citations appropriate for use in inline text. For example,

```
% \citet{hasselmo} investigated\dots
```

produces

Hasselmo, et al. (1995) investigated…

If you wish to load the `natbib` package with options, you may add the following before loading the `neurips_2025` package:

```
\PassOptionsToPackage{options}{natbib}
```

If `natbib` clashes with another package you load, you can add the optional argument `nonatbib` when loading the style file:

```
\usepackage[nonatbib]{neurips_2025}
```

As submission is double blind, refer to your own published work in the third person. That is, use "In the previous work of Jones et al. [4]," not "In our previous work [4]." If you cite your other papers that are not widely available (e.g., a journal paper under review), use anonymous author names in the citation, e.g., an author of the form "A. Anonymous" and include a copy of the anonymized paper in the supplementary material.

6.2 Footnotes

Footnotes should be used sparingly. If you do require a footnote, indicate footnotes with a number[2] in the text. Place the footnotes at the bottom of the page on which they appear. Precede the footnote with a horizontal rule of 2 inches (12 picas).

Note that footnotes are properly typeset *after* punctuation marks.[3]

6.3 Figures

All artwork must be neat, clean, and legible. Lines should be dark enough for purposes of reproduction. The figure number and caption always appear after the figure. Place one line space before the figure caption and one line space after the figure. The figure caption should be lower case (except for first word and proper nouns); figures are numbered consecutively.

You may use color figures. However, it is best for the figure captions and the paper body to be legible if the paper is printed in either black/white or in color.

[2] Sample of the first footnote.

[3] As in this example.

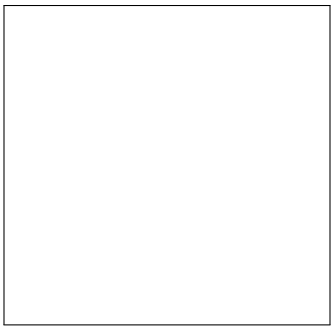

Figure 1: Sample figure caption.

Table 1: Sample table title

	Part	
Name	Description	Size (μm)
Dendrite	Input terminal	$\sim$100
Axon	Output terminal	$\sim$10
Soma	Cell body	up to 10^6

6.4 Tables

All tables must be centered, neat, clean and legible. The table number and title always appear before the table. See Table 1.

Place one line space before the table title, one line space after the table title, and one line space after the table. The table title must be lower case (except for first word and proper nouns); tables are numbered consecutively.

Note that publication-quality tables *do not contain vertical rules.* We strongly suggest the use of the `booktabs` package, which allows for typesetting high-quality, professional tables:

`https://www.ctan.org/pkg/booktabs`

This package was used to typeset Table 1.

6.5 Math

Note that display math in bare TeX commands will not create correct line numbers for submission. Please use LaTeX (or AMSTeX) commands for unnumbered display math. (You really shouldn't be using $$ anyway; see `https://tex.stackexchange.com/questions/503/why-is-preferable-to` and `https://tex.stackexchange.com/questions/40492/what-are-the-differences-between-align-equation-and-displaymath` for more information.)

6.6 Final instructions

Do not change any aspects of the formatting parameters in the style files. In particular, do not modify the width or length of the rectangle the text should fit into, and do not change font sizes (except perhaps in the **References** section; see below). Please note that pages should be numbered.

7 Preparing PDF files

Please prepare submission files with paper size "US Letter," and not, for example, "A4."

Fonts were the main cause of problems in the past years. Your PDF file must only contain Type 1 or Embedded TrueType fonts. Here are a few instructions to achieve this.

- You should directly generate PDF files using `pdflatex`.
- You can check which fonts a PDF files uses. In Acrobat Reader, select the menu Files>Document Properties>Fonts and select Show All Fonts. You can also use the program `pdffonts` which comes with `xpdf` and is available out-of-the-box on most Linux machines.
- `xfig` "patterned" shapes are implemented with bitmap fonts. Use "solid" shapes instead.
- The `\bbold` package almost always uses bitmap fonts. You should use the equivalent AMS Fonts:

  ```
  \usepackage{amsfonts}
  ```

  followed by, e.g., `\mathbb{R}`, `\mathbb{N}`, or `\mathbb{C}` for $\mathbb{R}$, $\mathbb{N}$ or $\mathbb{C}$. You can also use the following workaround for reals, natural and complex:

  ```
  \newcommand{\RR}{I\!\!R} %real numbers
  \newcommand{\Nat}{I\!\!N} %natural numbers
  \newcommand{\CC}{I\!\!\!\!C} %complex numbers
  ```

  Note that `amsfonts` is automatically loaded by the `amssymb` package.

If your file contains type 3 fonts or non embedded TrueType fonts, we will ask you to fix it.

7.1 Margins in LaTeX

Most of the margin problems come from figures positioned by hand using `\special` or other commands. We suggest using the command `\includegraphics` from the `graphicx` package. Always specify the figure width as a multiple of the line width as in the example below:

```
\usepackage[pdftex]{graphicx} ...
\includegraphics[width=0.8\linewidth]{myfile.pdf}
```

See Section 4.4 in the graphics bundle documentation (`http://mirrors.ctan.org/macros/latex/required/graphics/grfguide.pdf`)

A number of width problems arise when LaTeX cannot properly hyphenate a line. Please give LaTeX hyphenation hints using the `\-` command when necessary.

Acknowledgments and Disclosure of Funding

Use unnumbered first level headings for the acknowledgments. All acknowledgments go at the end of the paper before the list of references. Moreover, you are required to declare funding (financial activities supporting the submitted work) and competing interests (related financial activities outside the submitted work). More information about this disclosure can be found at: `https://neurips.cc/Conferences/2025/PaperInformation/FundingDisclosure`.

Do **not** include this section in the anonymized submission, only in the final paper. You can use the `ack` environment provided in the style file to automatically hide this section in the anonymized submission.

References

References follow the acknowledgments in the camera-ready paper. Use unnumbered first-level heading for the references. Any choice of citation style is acceptable as long as you are consistent. It is permissible to reduce the font size to `small` (9 point) when listing the references. Note that the Reference section does not count towards the page limit.

[1] Alexander, J.A. & Mozer, M.C. (1995) Template-based algorithms for connectionist rule extraction. In G. Tesauro, D.S. Touretzky and T.K. Leen (eds.), *Advances in Neural Information Processing Systems 7*, pp. 609–616. Cambridge, MA: MIT Press.

[2] Bower, J.M. & Beeman, D. (1995) *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* New York: TELOS/Springer–Verlag.

[3] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region CA3. *Journal of Neuroscience* **15**(7):5249-5262.

A Technical Appendices and Supplementary Material

Technical appendices with additional results, figures, graphs and proofs may be submitted with the paper submission before the full submission deadline (see above), or as a separate PDF in the ZIP file below before the supplementary material deadline. There is no page limit for the technical appendices.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

References

Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. On the worst prompt performance of large language models. In *Advances in Neural Information Processing Systems*, 2024.

Chao Lei et al. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. BERTScore: Evaluating text generation with BERT. In *International Conference on Learning Representations*, 2020.