Struggling at the Start: Structural Causes of Decoding Difficulty in Code Generation

Anonymous Authors

Abstract

Large Language Models (LLMs) demonstrate strong performance in code generation, yet generation errors remain prevalent. Prior analyses primarily focus on final outputs and aggregate performance metrics, with limited examination of the decoding process that gives rise to these outputs. To address this, we conduct a systematic token-level comparison between natural language and code generation, and identify consistent differences in their decoding behaviors. We find that decoding difficulty in code is not uniformly distributed, but is concentrated at structurally critical positions, particularly at line-initial tokens. Based on these observations, we propose an interpretation in which decoding difficulty is associated with increased predictive uncertainty at high-level structural decision points during generation. To probe this interpretation, we introduce a lightweight prompt-level intervention that provides structural guidance, enabling a controlled diagnostic analysis without modifying LLMs or decoding strategies. Experiments on HumanEval and MBPP show that this intervention consistently reduces predictive entropy at line-initial positions, highlighting the localized nature of decoding difficulty and motivating future work on structure-aware code generation.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

Do, Not, Use, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:

Anonymous Authors. 2026. Struggling at the Start: Structural Causes of Decoding Difficulty in Code Generation . In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide range of natural language processing tasks [8] and software engineering applications, including code generation [2, 7]. Despite these advances, code generated by LLMs remains brittle: minor decoding errors can easily cascade into syntactic or semantic failures [11, 14], posing a challenge for reliable deployment in real-world programming scenarios.

Prior analyses primarily focus on final outputs and aggregate performance metrics, with limited examination of the decoding process. In autoregressive generation, decoding determines how an LLM incrementally selects each next token, directly shaping the correctness and coherence of the generated output. A key source of syntactic or semantic failures manifests during the decoding process, specifically in how the model makes token-level decisions. Compared to natural language, programming languages impose strict syntactic and semantic constraints, which fundamentally reshape the space of valid token-level continuations during decoding. In this work, we begin by conducting a systematic token-level comparison between these two settings. Using metrics such as loss variance, perplexity, and distributional skewness, we observe a consistent contrast: while the distribution of predictive loss in natural language generation is relatively dispersed, code generation exhibits highly localized instability, where most tokens are predicted with high confidence but a small subset incur substantially large loss. This heavy-tailed loss distribution suggests that decoding difficulty in code is concentrated at specific positions.

Motivated by this observation, we identify and characterize a class of *difficult tokens*—tokens that consistently exhibit elevated predictive loss and uncertainty during code generation. We find that these tokens are not randomly distributed. Instead, they cluster at structurally critical locations, most notably at the first semantically meaningful token of a line. Such line-initial positions often coincide with points where the model must commit to high-level structural choices, such as control flow decisions, semantic intent, or the introduction of long-range dependencies. This empirical pattern suggests a candidate causal mechanism: unresolved structural alternatives positions contribute to increased decoding difficulty.

To examine this hypothesis, we recast the difficult-token phenomenon as a problem of causal diagnosis. Rather than modifying the model architecture or decoding algorithm, we introduce a targeted prompt-level diagnostic intervention that encourages the model to consider line-level structure prior to generation. Through this intervention, we observe a consistent reduction in predictive entropy at line-initial positions, suggesting that structural uncertainty at these positions is effectively alleviated. These findings are consistent with an interpretation that decoding difficulty at line starts is closely related to structural ambiguity.

Beyond the proposed intervention, this causal perspective offers a interpretation of structure-aware code generation methods, such as SKCoder[10]. By explicitly generating high-level structural sketches prior to token-level realization, such approaches can be viewed as applying stronger interventions on similar structural decision points. Although motivated by performance optimization, their empirical success provides complementary evidence consistent with the causal diagnostic view advanced in this work.

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In summary, this paper makes the following contributions:

- We present a systematic token-level analysis comparing decoding behavior in natural language and code generation, revealing localized instability unique to code.
- We identify line-initial structural positions as key locations where predictive uncertainty concentrates, and hypothesize that unresolved structural decisions at these positions give rise to elevated decoding difficulty.
- We introduce a lightweight, diagnostic prompt-level intervention to test this hypothesis and motivating future structure-aware generation.

2 Differences in decoding in both natural language and code generation

Before analyzing decoding behavior in code generation, a central question is whether the decoding process for code exhibits properties that are distinct from those of natural language generation. To investigate this question, we begin by systematically comparing the decoding behavior of LLMs in natural language and code generation.

2.1 Metrics

To analyze the decoding behaviors of large language models (LLMs) in both code generation and natural language generation tasks, we construct different quantitative metrics to characterize the distributional properties of the decoding process. We adopt the token-level log loss[15] as the core metric to evaluate LLM performance. Given a generated sequence $x_1, x_2, \ldots, x_n$, LLMs predict each token in an autoregressive manner, the token-level log loss is defined as: $L_i = -\log P(x_i \mid x_1, x_2, \ldots, x_{i-1})$ This metric directly reflects the model's prediction confidence at each step. In the information theory[13], log loss represents the self-information of the predicted token. To capture and compare the statistical properties of loss value during the decoding process, we use the following statistical metrics over token log losses:

Standard Deviation (σ). Measures the variance of log losses within a sequence:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(L_i - \mu)^2}, \quad \text{where} \quad \mu = \frac{1}{n}\sum_{i=1}^{n} L_i \tag{1}$$

A low standard deviation indicates stable prediction confidence across tokens.

Skewness (γ). measures the asymmetry of the log loss distribution:

$$\gamma = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{L_i - \mu}{\sigma}\right)^3 \tag{2}$$

Positive skew indicates that most tokens are predicted with low loss but a few are highly uncertain, while negative skew suggests the reverse.

Perplexity (PPL). is defined as the exponential of the average log loss, perplexity offers a global summary of model prediction confidence:

$$\text{PPL} = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log P(x_i \mid x_1, \ldots, x_{i-1})\right) \tag{3}$$

Lower perplexity implies higher overall certainty and fluency in generation.

Table 1: Experimental Evaluation Results on Both Datasets

LLM	Dataset	Language	σ	γ	PPL
DeepSeek	$\mathcal{D}_H$	code	1.52	3.62	1.60
	$\mathcal{D}_H$	text	1.61	2.67	2.10
	$\mathcal{D}_A$	code	1.63	3.41	1.85
	$\mathcal{D}_A$	text	1.98	1.94	3.06
Qwen	$\mathcal{D}_H$	code	0.82	3.35	1.31
	$\mathcal{D}_H$	text	1.05	2.84	1.58
	$\mathcal{D}_A$	code	1.58	3.55	1.87
	$\mathcal{D}_A$	text	1.71	2.45	2.35
CodeLlama	$\mathcal{D}_H$	code	1.52	3.98	1.56
	$\mathcal{D}_H$	text	1.55	2.62	2.22
	$\mathcal{D}_A$	code	1.46	4.44	1.61
	$\mathcal{D}_A$	text	1.92	1.97	2.97

2.1.1 Dataset Construction. We investigates the differences in decoding behaviors between code and natural language generation using three LLMs: DeepSeek-Coder-7B[3], Qwen-Coder-7B[5], and CodeLlama-13B[12]. To support this analysis, we construct paired datasets comprising program code and natural language:

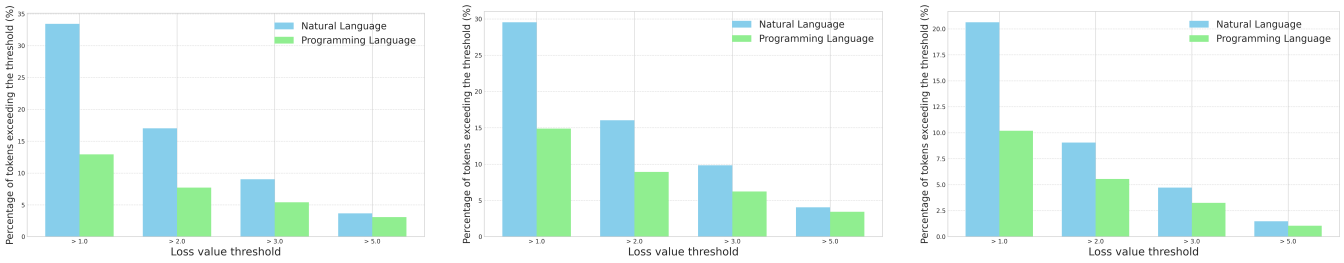
Manually Constructed Dataset ($\mathcal{D}_H$). We build a high-quality dataset based on HumanEval [2] and MBPP [1]. From each dataset, we randomly select 20 problems. We manually write corresponding natural language descriptions based on the code implementations. These descriptions are written and cross-checked by two programmers with at least two years of software development experience, ensuring logical consistency and semantic clarity.

Automatically Constructed Dataset ($\mathcal{D}_A$). We also constructed a large automatically-generated dataset based on the Python subset of CodeSearchNet[6]. We used DeepSeek-Coder-V2[3] to generate natural language descriptions. This process resulted in a dataset of 21,920 (text, code) pairs, reflecting the complexity of real-world programming.

2.2 Experimental Results and Analysis

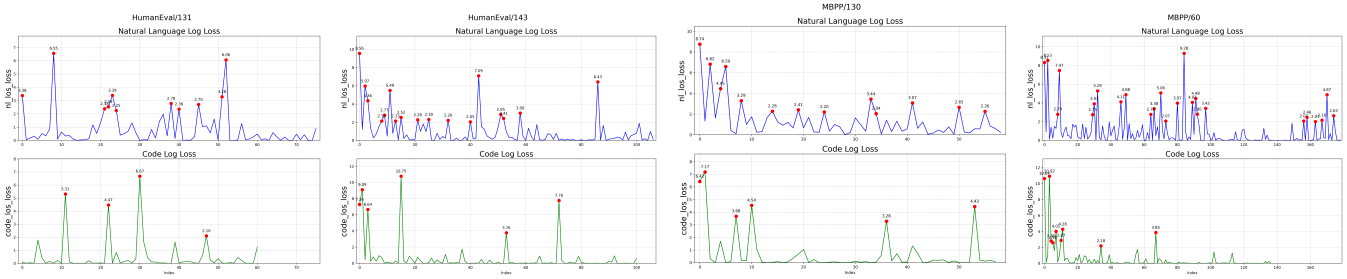
We calculate the average of evaluation metrics across all samples and show the results in Table 1. We derive the following key observations:

Code generation shows lower variance in prediction loss than text generation On $\mathcal{D}_H$, the average standard deviation for code is 1.28, and 1.40 (+9.3%) for text. On the $\mathcal{D}_A$, it rises to 1.55, and 1.87 (+20.6%). Code generation has less high-loss tokens and higher skewness than text generation. On $\mathcal{D}_H$, average skewness for code is 3.66, and 2.71 for text (-26.3%); on $\mathcal{D}_A$, it's 3.80, and 2.12 (-44.2%).



(a) CodeLlama (b) DeepSeek-Coder (c) Qwen-Coder

Figure 1: Statistical results of high-loss token proportions on dataset D_H



(a) HumanEval/131 (b) HumanEval/143 (c) MBPP/130 (d) MBPP/60

Figure 2: Illustration of loss value changes in natural language and programming language decoding processes

This indicates that token losses in code are more sparsely and unevenly distributed than in text. Code generation has lower perplexity than text generation, indicating lower prediction uncertainty. On $\mathcal{D}_H$, average perplexity is 1.49 for code and 1.96 for text (+31.5%). On $\mathcal{D}_A$, it's 1.77, and 2.79 (+34%).

Moreover, we compare the proportions of high-loss tokens on dataset $\mathcal{D}_H$ and show the results in Figure 1. The results reveal clear differences: natural language consistently has a higher percentage of tokens exceeding loss thresholds than code. For instance, at a loss threshold of 1.0, 33.5% of natural language tokens exceed it, compared to only 12.9% for code. Although the proportions decrease with higher thresholds, natural language remains higher throughout. This difference reflects the stricter syntax of programming languages. Natural language's flexibility allows multiple valid outputs, leading to greater prediction volatility and more high-loss tokens.

We also visualizes loss value changes of LLMs during natural language and code decoding, shown in Figure 2. The figure displays prediction loss trends for four samples, with blue curves for natural language and green for code. Tokens with high losses (> 2) are marked in red and annotated. More experimental results are shown in Appendix A.1 The results show distinct peak patterns: code generation features few sharp high-loss peaks amid mostly near-zero losses, indicating key decision points in code generation.

3 Difficult tokens analysis in code generation

To further understand the prediction difficulties of LLMs in the decoding process of code generation tasks. We define the difficult tokens in code generation and conduct extensive analysis.

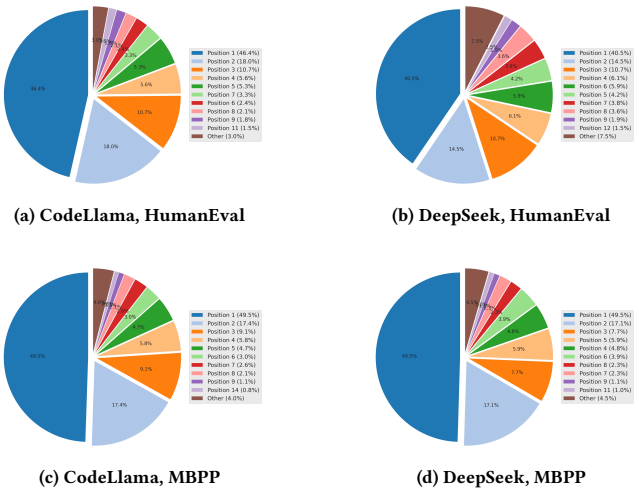
3.1 Difficult tokens definition

To evaluate token-level model performance in code generation, we propose the Predictive Difficulty (PD) metric, which measures how hard a token is to predict based on its loss rank: $\mathrm{PD}(t_i) = \frac{\mathrm{Rank}(t_i)}{N}$ where t_i is the i-th token, $\mathrm{Rank}(t_i)$ is its rank by prediction loss (ascending), and N is the total number of tokens. Inspired by prior work [9], we define difficult tokens as those with PD above a threshold H:

$$\text{Difficult Token}(t_i) = \begin{cases} 1 & \text{if } \mathrm{PD}(t_i) > H \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

3.2 Experimental Results and Analysis

To analyze where difficult tokens occur during code generation, we evaluate DeepSeek-Coder and CodeLlama on HumanEval, MBPP, and APPS[4]. We assign position indices within each code line, marking indentation as Position 0 and the first meaningful token as Position 1. Across all datasets and models, Position 1 consistently contains the most difficult tokens (shown in Figure 3). On MBPP, Position 1 accounts for 49.5% of difficult tokens, far exceeding the second-most difficult position (17.1%), showing a relative increase



(a) CodeLlama, HumanEval (b) DeepSeek, HumanEval

(c) CodeLlama, MBPP (d) DeepSeek, MBPP

Figure 3: Threshold Analysis of the Proportion of Difficult Tokens, $H = 0.95$

of over 180%. Similar patterns are observed on HumanEval and APPS (shown in Appendix A.3), where Position 1 holds over 40+% of difficult tokens. We visualize difficult tokens in red and provide representative examples in Appendix A.4.

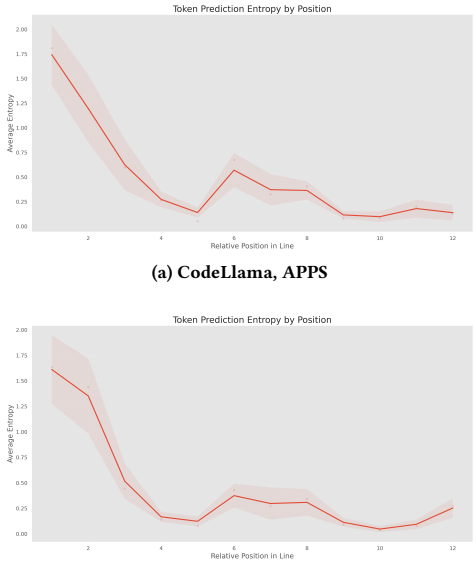
This behavior reflects the cognitive and structural significance of line-initial tokens: they often require reasoning over previous context, understanding the broader structure of the code, and anticipating future development in the code logic. In contrast, tokens appearing later within a line are more likely to rely on local syntax or autocomplete-style patterns, resulting in a flatter difficulty distribution. We demonstrate the distribution of difficult tokens in Appendix A.4.

To further analyze why line-initial tokens incur higher decoding loss, we examine whether this difficulty arises from overconfident but incorrect predictions, or from increased predictive uncertainty at these positions. To this end, we visualize the entropy of model predictions across different token positions. As depicted in Figure 4, we observe that predictive entropy consistently peaks at Position 1 across datasets and models, indicating elevated uncertainty at line beginnings (additional results are provided in Appendix A.2).

These findings suggest that the high decoding loss observed at line-initial tokens is primarily associated with increased uncertainty rather than confident misprediction. This further supports the view that decoding at the start of a code line involves more complex reasoning and structural decision-making, leading to greater model uncertainty and motivating dedicated strategies for accurate generation.

4 Targeted Prompt-Level Intervention

The observational analyses in the previous sections reveal a strong and consistent association between line-initial positions and elevated decoding difficulty. However, correlation alone is insufficient to establish causal relevance. To move beyond correlation, we seek to assess whether this association reflects a causal relationship



(a) CodeLlama, APPS

(b) DeepSeek-Coder, APPS

Figure 4: Uncertainty of different positions (APPS)

rather than a coincidental alignment. In this section, we introduce a targeted prompt-level intervention that injects lightweight structural guidance, and examine whether such guidance reduces decoding uncertainty in code generation, particularly at line-initial structural decision points.

4.1 Intervention Design

The intervention is designed to manipulate the hypothesized causal factor—*line-initial structural uncertainty*—while keeping all other aspects of generation unchanged. Concretely, we prepend a lightweight structural guidance to the original prompt:

> *"Generate the following code carefully. Before starting, internally determine the structure of the code. Output only valid code. "*

This intervention has three important properties. First, it does not introduce task-specific hints, intermediate solutions, or external supervision. Second, it leaves the decoding strategy unchanged. Third, it explicitly targets the structural decision boundary at line starts, which our prior analysis identifies as a key locus of decoding uncertainty.

From a causal perspective, the intervention can be viewed as a controlled manipulation,

$$\texttt{do(StructuralGuidance = 1)},$$

where the outcome of interest is the model's predictive uncertainty, which is used as a token-level proxy for decoding difficulty and is measured via entropy under teacher forcing at different token positions.

4.2 Experimental Setup

We evaluate the proposed intervention on two standard code generation benchmarks, HumanEval [2] and MBPP [1]. Experiments are

Table 2: Effect of prompt-level structural guidance on predictive entropy. Results are reported separately for line-initial tokens (Position 1) and non-line-initial tokens. Lower entropy indicates reduced predictive uncertainty.

Model	Dataset	Position 1 (Line-Initial)			Non-Position 1		
		Base	Interv.	Δ	Base	Interv.	Δ
Qwen	HumanEval	0.9450	0.6765	0.2685	0.2669	0.1649	0.1020
Qwen	MBPP	0.9718	0.6239	0.3479	0.4187	0.2827	0.1360
DeepSeek	HumanEval	0.7832	0.6936	0.0896	0.1941	0.1924	0.0017
DeepSeek	MBPP	0.8817	0.8417	0.0400	0.3426	0.3396	0.0030

conducted using two representative code LLMs, Qwen2.5-Coder-7B and DeepSeek-Coder-7B. For each task and model, we compare two conditions: (i) the original prompt (baseline), and (ii) the prompt augmented with structural guidance (intervention).

For both conditions, we compute token-level predictive entropy for **line-initial tokens (Position 1)** and **non-line-initial tokens**. All other experimental settings are kept identical across conditions. This controlled setup ensures that any observed differences in uncertainty can be attributed to the intervention itself.

4.3 Experimental Results

Table 2 summarizes the effect of the prompt-level intervention across models and datasets. Across all four settings, we observe a consistent reduction in predictive entropy at line-initial positions under the intervention condition. For Qwen2.5-Coder, the reduction is substantial, reaching 0.27 on HumanEval and 0.35 on MBPP. DeepSeek-Coder exhibits a similar but more moderate pattern, with smaller yet consistent reductions at line starts.

Crucially, the entropy reduction is localized. For non-line-initial tokens, the corresponding reductions are markedly smaller despite the intervention being applied globally at the prompt level. This contrast indicates that the intervention does not uniformly reshape the model's output distribution.

The strong localization of entropy reduction provides causal diagnostic evidence that the intervention primarily operates by alleviating uncertainty at *structural decision boundaries*, rather than by globally smoothing token predictions. Because the decoding strategy and evaluation targets remain unchanged, and because uncertainty reductions concentrate specifically at line-initial positions, the observed effect cannot be explained by generic calibration or prompt-induced regularization. Instead, these results support the causal hypothesis that line-initial positions correspond to latent planning points where the model must commit to a high-level structural continuation.

4.4 Discussion

Taken together, these results provide causal diagnostic evidence that unresolved structural decisions at line starts contribute substantially to elevated decoding difficulty in code generation. By intervening on the availability of structural guidance at these positions, we are able to systematically reduce decoding difficulty of code generation with LLMs. These findings strengthen our central claim that line-initial structural ambiguity plays a key role in shaping decoding difficulty, and help explain why structure-aware generation strategies can be effective in practice. We further connect our findings to structure-first generation methods such as SKCoder[10], whose empirical success provides complementary, external validation of the same causal mechanism.

5 Conclusion

In this work, we provide a causal diagnosis of decoding difficulty in code generation. By comparing token-level decoding behaviors between natural language and code, we show that doecoding difficulty in code generation is highly localized, which is concentrated at line-initial structural positions. We interpret these positions as high-level planning and decision points that induce elevated predictive difficulty.

To test this hypothesis, we introduce a lightweight, diagnostic prompt-level intervention that introducing strutural guidance during decoding, and demonstrate its effectiveness in lowering uncertainty at these positions.

6 Future Work

This work opens several promising directions for future research:

First, our intervention is deliberately lightweight and diagnostic. An important direction is to integrate causal insights into more powerful generation strategies. For example, combining minimal structural guidance with structure-first or plan-and-execute paradigms may yield decoding methods that are both interpretable and effective. Our findings provide a causal explanation for why such approaches work, but their systematic design remains an open problem.

Second, future work may explore causal modeling beyond prompt-level interventions. This includes learning causal abstractions from model hidden states, constructing explicit causal graphs over structural and semantic variables, or performing controlled interventions at the representation or decoding-policy level. Such approaches could deepen our understanding of how internal model dynamics give rise to observable decoding behavior.

Finally, while this study focuses on code generation, the proposed causal diagnostic framework may extend to other structured generation tasks, such as mathematical reasoning, formal proof generation, and program synthesis. Investigating whether similar structure-induced uncertainty arises in these domains would further clarify the role of planning and structure in foundation model behavior.

We hope this work motivates further integration of causal reasoning into the analysis and design of decoding methods, advancing the interpretability and trustworthiness of large language models.

References

[1] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). https://arxiv.org/abs/2108.07732 arXiv: 2108.07732.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé 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, Joshua 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. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). https://arxiv.org/abs/2107.03374 arXiv: 2107.03374.

[3] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming - The Rise of Code Intelligence. *CoRR* abs/2401.14196 (2024). doi:10.48550/ARXIV.2401.14196 arXiv: 2401.14196.

[4] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[5] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. *CoRR* abs/2409.12186 (2024). doi:10.48550/ARXIV.2409.12186 arXiv: 2409.12186.

[6] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[7] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[8] Pranjal Kumar. 2024. Large language models (LLMs): survey, technical frameworks, and future challenges. *Artificial Intelligence Review* 57, 10 (2024), 260.

[9] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[10] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. SkCoder: A Sketch-based Approach for Automatic Code Generation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2124–2135. doi:10.1109/ICSE48619.2023.00179

[11] Sherlock A Licorish, Ansh Bajpai, Chetan Arora, Fanyu Wang, and Kla Tantithamthavorn. 2025. Comparing Human and LLM Generated Code: The Jury is Still Out! *arXiv preprint arXiv:2501.16857* (2025).

[12] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). doi:10.48550/ARXIV.2308.12950 arXiv: 2308.12950.

[13] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[14] Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, and Giuliano Antoniol. 2025. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering* 30, 3 (2025), 1–48.

[15] Vladimir Vovk. 2015. The fundamental nature of the log loss function. *Fields of logic and computation II: Essays dedicated To Yuri Gurevich on the Occasion of His 75th Birthday* (2015), 307–318.

A Appendix

A.1 Illustration of loss value changes

Illustration of loss value changes on examples from MBPP datasets. The results are shown in Figure 5.

A.2 Uncertainty Estimation

Uncertainty Estimation of positions in code lines on HumanEval and MBPP datasets. The results are shown in Figure 6.

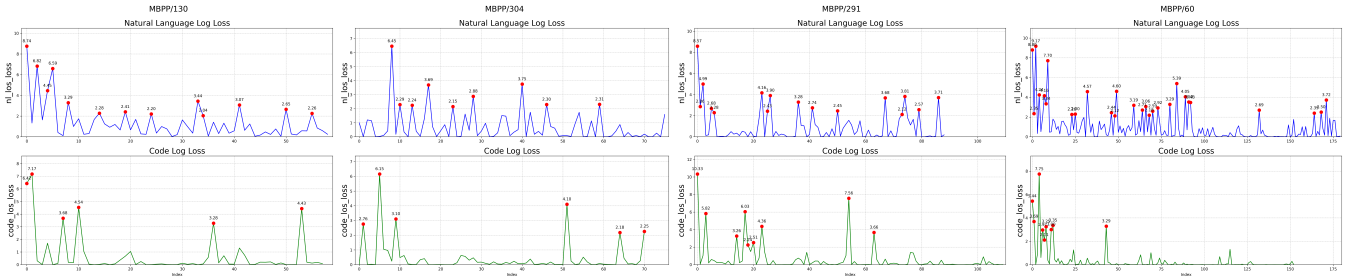
A.3 Proportion of Difficult Tokens

Proportion of Difficult Tokens of different positions in code lines on HumanEval and MBPP datasets. We also report the results under $H = 0.85$. The results are shown in Figure 7.

A.4 Example of Difficult Tokens

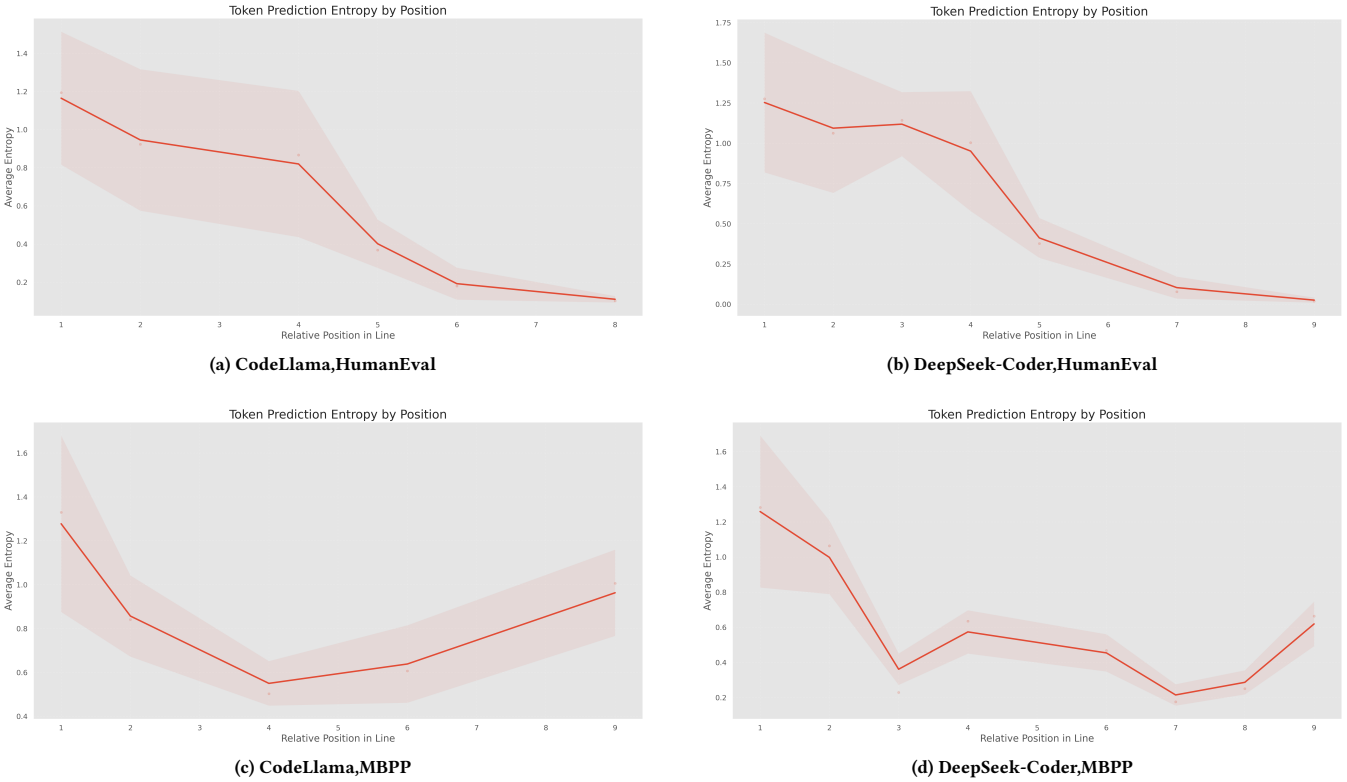
Example of Difficult Tokens in code snippet. The results are shown in Figure 8.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009



(a) MBPP/130 (b) MBPP/291 (c) MBPP/130 (d) MBPP/291

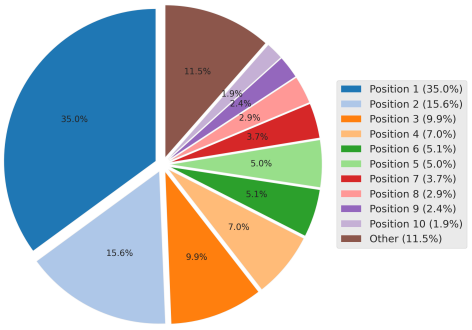
Figure 5: Illustration of loss value changes in natural language and programming language decoding processes



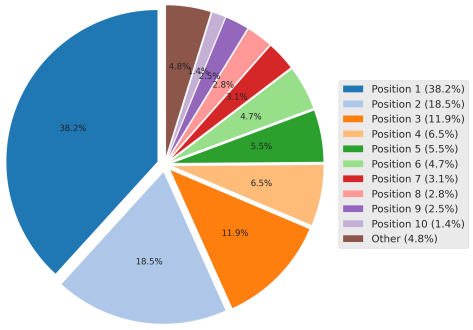
(a) CodeLlama,HumanEval (b) DeepSeek-Coder,HumanEval

(c) CodeLlama,MBPP (d) DeepSeek-Coder,MBPP

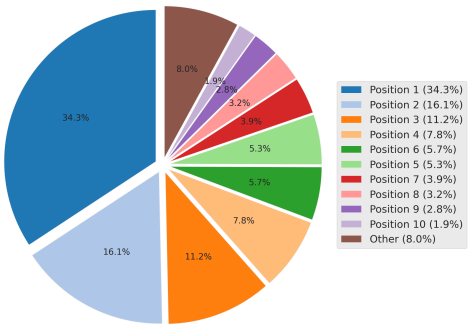
Figure 6: Uncertainty of different positions



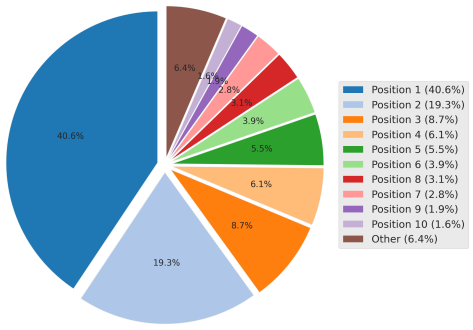
(a) CodeLlama,APPS,H=0.85



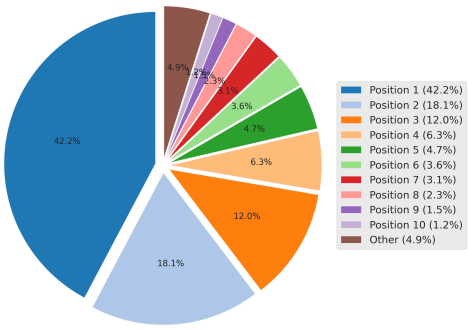
(b) CodeLlama,HumanEval,H=0.85



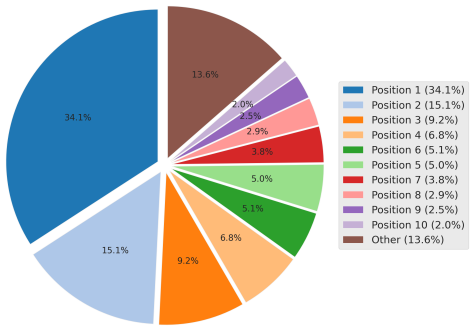
(c) DeepSeek-Coder,HumanEval,H=0.85



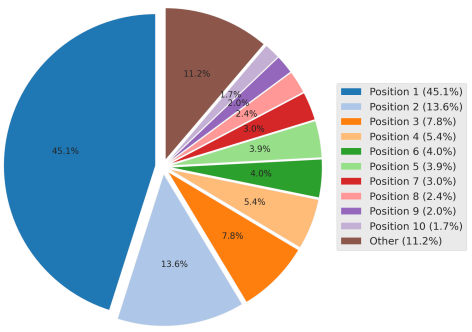
(d) DeepSeek-Coder, MBPP,H=0.85



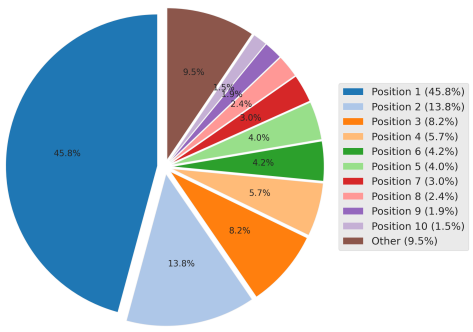
(e) CodeLlama, MBPP, H=0.85



(f) DeepSeek-Coder, APPS, H=0.85



(g) DeepSeek-Coder, APPS, H=0.95



(h) CodeLlama, APPS, H=0.95

Figure 7: Threshold-based analysis of the proportion of difficult tokens at the beginning of code lines.

```
scaler = StandardScaler()
standardized_data = scaler.fit_transform(data)
standardized_data_str = np.array2string(standardized_data)
encoded_data = base64.b64encode(standardized_data_str.encode('ascii')).decode('ascii')

return encoded_data
```

```
elements_series = pd.Series(elements)
count_series = elements_series.apply(lambda x: len(x))
data_dict = {'Element': elements_series, 'Count': count_series}
if include_index:
    data_dict['Index'] = np.arange(len(elements))
count_df = pd.DataFrame(data_dict)
if include_index:
    count_df = count_df[['Index', 'Element', 'Count']]  # Reordering columns to put 'Index' first
return count_df
```

```
strings = [''.join(random.choices(string.ascii_lowercase, k=string_length)) for _ in range(num_strings)]
characters = ''.join(strings)
character_counter = Counter(characters)
most_common_characters = character_counter.most_common()

return most_common_characters
```

```
X = df.drop(target_column, axis=1)
y = df[target_column]

model = RandomForestClassifier(random_state=42).fit(X, y)
feature_imp = pd.Series(model.feature_importances_, index=X.columns).sort_values(
    ascending=False
)
plt.figure(figsize=(10, 5))
ax = sns.barplot(x=feature_imp, y=feature_imp.index)
ax.set_xlabel("Feature Importance Score")
ax.set_ylabel("Features")
ax.set_title("Visualizing Important Features")
return model, ax
```

```
font = {'family': 'Arial'}
plt.rc('font', **font)  # Set the global font to Arial.
DIABETES = load_diabetes()
diabetes_df = pd.DataFrame(data=DIABETES.data, columns=DIABETES.feature_names)
pair_plot = sns.pairplot(diabetes_df)
return pair_plot.fig, diabetes_df
```

```
def find_poisoned_duration(self, time_series, duration):
    if not time_series:
        return 0
    total = 0
    for i in range(1, len(time_series)):
        total += min(time_series[i] - time_series[i - 1], duration)
    return total + duration
```

```
z_scores = zscore(data_matrix, axis=1)
feature_columns = ["Feature " + str(i + 1) for i in range(data_matrix.shape[1])]
df = pd.DataFrame(z_scores, columns=feature_columns)
df["Mean"] = df.mean(axis=1)
correlation_matrix = df.corr()
ax = sns.heatmap(correlation_matrix, annot=True, fmt=".2f")
return df, ax
```

Figure 8: Example of Difficult Tokens in code snippet