Efficient Hallucination Detection in Automatic Code Generation

Anonymous ACL submission

Abstract

Large language models (LLMs) frequently generate source code that appears plausible yet contains hallucinations that lead to test failures. While uncertainty quantification (UQ) methods have shown promise for hallucination detection in natural language, their effectiveness in the code generation setting remains largely unexplored. In this work, we investigate the performance of state-of-the-art UQ methods for hallucination detection in source code generation and propose an efficient and effective training-based approach. We develop a diff-based pipeline to construct a code dataset annotated with line-level LLM hallucinations, enabling systematic benchmarking of hallucination detection methods. Using this pipeline, we build a large-scale annotated dataset and train a lightweight Transformer-based hallucination detector that leverages LLM hidden states as input features. Experimental results across diverse code generation domains demonstrate that the detector substantially outperforms other existing approaches in line-level hallucination detection. We release the first publicly available dataset of line-level code hallucinations, along with the corresponding source code and trained hallucination detectors.

1 Introduction

Despite the impressive performance of Large Language Models (LLMs) in code generation, the code they produce is not safe from incorrect behavior (Sharma and David, 2025; Tian et al., 2025). For instance, Figure 1 illustrates a case where the generated code is syntactically valid yet fails to execute correctly or produces incorrect outputs. Such errors may be attributed to LLM hallucinations (Zhang et al., 2025). Uncertainty quantification (UQ) methods offer a promising way to identify unreliable and potentially hallucinated model outputs (Gal et al., 2016; Malinin and Gales, 2021). While hallucination detection has been studied extensively in natural language processing tasks (Kuhn et al., 2023; Fadeeva et al., 2023; Vashurin et al., 2025), its performance in source code generation remains underexplored, especially at fine-grained levels such as lines of code or tokens.

TASK:

Write a function to find the specified number of largest products from two given lists.

CORRECT SOLUTION:

```
1: def large_product(nums1, nums2, N):
2:     mul = []
3:     for i in range(len(nums1)):
4:         for j in range(len(nums2)):
5:             mul.append(nums1[i] * nums2[j])
6:     res = sorted(mul, reverse=True)[:N]
7:     return res
```

HALLUCINATED SOLUTION:

```
1: def large_product(nums1, nums2, N):
2:     res = []
3:     for i in range(N):
4:         res.append(max(nums1[i] * nums2[i]))
5:     return res
```

Figure 1: An example of hallucinations in code generation (HumanEval). Correct lines of the wrong solution are highlighted in green, while hallucinated lines are highlighted in red. Line 3 of the wrong solution is hallucinated because it incorrectly assumes the top N products can be derived from the first N indices. Line 4 of the wrong solution is hallucinated because it only considers products at the same index and ignores all other valid combinations.

In this work, we conduct a comprehensive evaluation of UQ methods on the task of hallucination detection in code generation. We develop a diff-based pipeline to construct a code dataset annotated

with line-level LLM hallucinations, enabling systematic benchmarking of hallucination detection methods. Using this pipeline, we build a large-scale annotated dataset and train a lightweight Transformer-based hallucination detector that leverages LLM hidden states as input features. Experimental results across diverse code generation domains demonstrate that the detector substantially outperforms other existing approaches in line-level hallucination detection. We release the first publicly available dataset of line-level code hallucinations, along with the corresponding source code and trained hallucination detectors. Summarizing, the study makes the following contributions:

1. **A data annotation pipeline for benchmarking** hallucination detection methods in code generation. The pipeline can be used both to construct training datasets for supervised hallucination detectors and to provide ground-truth line-level labels for evaluation using an arbitrary LLM as the code generator.
2. **An annotated dataset** of line-level hallucinations in source code.
3. **A supervised model** for hallucination detection in LLM-generated source code and a series of pre-trained hallucination detectors for several state-of-the-art open-source LLMs.
4. **An extensive empirical investigation** of unsupervised uncertainty quantification methods and supervised hallucination detectors across various LLMs, code generation domains (code synthesis and code repair), and code hallucination types. The empirical investigation also includes an analysis of the most impactful features for detecting code hallucinations.

2 Related Work

2.1 Hallucination Detection for Source Code

There are few methods for detecting hallucinations in generated code using execution results (Sharma and David, 2025; Ravuri and Amarasinghe, 2025; Tian et al., 2025; Valentin et al., 2025) or LLM internal values (Kotti et al., 2025; Somov and Tutubalina, 2025). These methods mostly focus on function-level hallucination detection; that is, whether the entire code snippet contains hallucinations. However, to the best of our knowledge, Collu-Bench (Jiang et al., 2024) is the only study that proposes methods for code hallucination detection at a granularity lower than that of a function. The estimated criteria for hallucination detection are sometimes used as a proxy metric of functional correctness (Sharma and David, 2025). In this work, we are the first to propose the line-level training-based approach to hallucination detection and demonstrate its effectiveness and efficiency. We also the first to benchmark a wide range of UQ baselines and supervised hallucination detectors on the code generation task.

2.2 Uncertainty Quantification Methods

2.2.1 Unsupervised Methods

Uncertainty quantification methods are often used to detect hallucinations in natural language texts (Fadeeva et al., 2023). Unsupervised approaches use token probabilities, perplexity, attention weights, and entropy to estimate uncertainty of the LLM in generated text. These methods do not require training data, but usually struggle to detect hallucinations well.

Maximum Sequence Probability (MSP) (Fadeeva et al., 2023) is one of the simplest methods to estimate the uncertainty of a sequence of tokens. The uncertainty of the LLM in the generated sequence of tokens is estimated as the negative sum of log probabilities for each token. The sum is negated to ensure that a lower probability corresponds to higher uncertainty.

Perplexity (Kotti et al., 2025) is a method similar to MSP. It also estimates LLM uncertainty as a negative sum of log probabilities, but normalizes it by the length of the sequence. As in the MSP method, perplexity ensures that the lower the sequence probability, the higher the uncertainty.

Attention Score (Sriramanan et al., 2024) calculates uncertainty based on how much attention the LLM pays to the generated tokens. If in its middle layer the LLM pays strong attention to the tokens, uncertainty is low. Otherwise, uncertainty is high.

Maximum Token Entropy (MTE) (Fadeeva et al., 2023; Somov and Tutubalina, 2025) defines LLM uncertainty as the maximum token entropy within a generated sequence of tokens.

2.2.2 Supervised Methods

In contrast to unsupervised UQ methods, supervised ones provide improved quality of uncertainty quantification based on internal values of the LLM.

SAPLMA (Azaria and Mitchell, 2023) uses hidden states (activations) of the intermediate layers of LLM to predict token uncertainties using a fully-connected neural network.

Dataset	Split	Total Samples	Total Lines	Hallucinated Lines	Correct Lines
Mixed (Code synthesis & Code repair)	Train	653	4359	805 (18%)	3554 (82%)
	Validation	164	1175	269 (23%)	906 (77%)
	Test	205	1359	275 (20%)	1084 (80%)
	Total	1022	6893	1349 (20%)	5544 (80%)
Code synthesis	Train	295	2543	404 (16%)	2139 (84%)
	Validation	74	639	104 (16%)	535 (84%)
	Test	93	813	134 (16%)	679 (84%)
	Total	462	3995	642 (16%)	3353 (84%)
Code repair	Train	358	1843	451 (24%)	1392 (76%)
	Validation	90	452	126 (28%)	326 (72%)
	Test	90	452	126 (28%)	326 (72%)
	Total	538	2747	703 (26%)	2044 (74%)

Table 1: Statistics of the collected datasets of line-level code hallucinations of DeepSeek-Coder 6.7B Instruct. The dataset consists of subsets specific to the code generation domains; the subset names are provided in the corresponding column. Each dataset contains an equal number of hallucinated and correct samples. The *Hallucinated Lines* and *Correct Lines* columns provide percentage information relative to the *Total Lines* column in parentheses.

Factoscope (He et al., 2024) proposes to leverage a complex feature vector obtained for each token to predict uncertainties. The feature vector consists of four types of LLM inner states: 1) hidden states from selected layers, 2) logits of top-m tokens from each layer, 3) cosine similarities between top token embeddings from adjacent layers, and 4) inverse token ranks across layers.

Lookback Lens (Chuang et al., 2024) predicts line uncertainties using a feature vector of lookback ratios derived from the LLM attention maps, and trains a logistic regression on these features.

Uncertainty Head (Shelmanov et al., 2025) is a neural network that consists of Transformer Encoder (Vaswani et al., 2017) and a fully-connected classifier. It may take a various set of features, including hidden states, token probabilities, attention maps, and produce uncertainty values.

3 Dataset Construction

3.1 Base Dataset

We considered Collu-Bench (Jiang et al., 2024) as the base dataset. It consists of hallucinated and ground-truth source code solutions to the problems from MBPP (Austin et al., 2021), HumanEval (Chen, 2021), HumanEval Java (Silva and Li, 2024), Defects4J (Just et al., 2014), and SWE-bench (Jimenez et al., 2023). The code was generated by DeepSeek-Coder (Guo et al., 2024), CodeLlama (Roziere et al., 2023), StarCoder (Lozhkov et al., 2024), and Llama 3 (Grattafiori et al., 2024) models of various sizes. The examples of the prompts are provided in Appendix I. The dataset was chosen since it is the only publicly available dataset with labelled hallucinations in the generated code. However, we consider the labelling of hallucinations in Collu-Bench lacking informativeness since all the source code tokens are labelled from a certain index onward.

3.2 Hallucination Detection Granularity

In this work, we propose detecting hallucinations in source code at the *line* (or *statement*) level. We adopt this granularity because line-level analysis aligns naturally with developers' workflows, precisely identifying problematic code in a clear and interpretable manner. While token-level detection could offer finer-grained localization, it often produces outputs that are harder to interpret and less practical for real-world use.

3.3 Code Hallucinations Labelling Algorithm

The crucial problem of the construction of the dataset of line hallucinations is labelling. We propose to label line hallucinations using `diff` utility according to the following algorithm:

1. Tokenize hallucinated and ground-truth code snippets into lines;
2. Compute diff between lines of hallucinated and ground-truth code snippets;
3. Label all the lines that should be removed from the hallucinated code snippet (marked as `"-"` in the diff) as hallucinating;
4. Label all the first lines that come right after lines addition (marked as `"+"` in the diff) as hallucinating. If there are line removals right before line additions, label the first line as correct.

Following the algorithm, we composed labelled datasets for DeepSeek-Coder, CodeLlama, and Llama LLMs. To the best of our knowledge, this study is the first one that provides a publicly available dataset of labelled hallucinating lines of LLM-generated code. The examples of line-level code hallucination labels are provided in Figure 1 for code generation task and in Figure 14 of Appendix K for code repair task.

The main limitation of the proposed labeling algorithm is its sensitivity to lexical differences in diffs. For instance, lines that implement the same functionality but use different variable names may be incorrectly labeled as hallucinations. This limitation could potentially be mitigated through source-code normalization or LLM-assisted labeling. Since Collu-Bench already provides normalized code, we do not apply additional normalization in our experiments.

3.4 Hallucination Type Labeling

To analyze the relative detection difficulty across different hallucination categories, we additionally annotated the dataset with hallucination type labels. The annotation scheme is inspired by the taxonomy introduced in CodeHaluEval (Tian et al., 2025), which categorizes hallucinated code into eight classes. We prompt a classifier LLM with pairs of hallucinated and ground-truth code, detailed descriptions of the hallucination categories, and execution feedback for the hallucinated code. The task of the classifier is to select the most appropriate hallucination class. To improve classification reliability, we use three different LLMs and aggregate the final class label via majority voting. The classifier LLMs were DeepSeek-Coder 33B[1], QwQ 32B[2], and CodeLlama 34B[3]. The template of the prompt is provided in Appendix 13.

4 Supervised Hallucination Detection

4.1 Input Features

Hidden states (Azaria and Mitchell, 2023; He et al., 2024; Shelmanov et al., 2025) are the outputs of a l hidden layer for a specific token t:

$$F_{\text{hs}}(t) = h_l(t) \tag{1}$$

We use hidden states collected from multiple hidden layers of the LLM.

Attention weights (Shelmanov et al., 2025) is a vector containing raw attention weights from token t_i to its k immediate predecessors across all heads and layers:

$$F_{\text{aw}}(t_i) = \left\{ \alpha_{i,i-j}^{q,l} \right\}_{j=1}^{k} \tag{2}$$

for all $q \in \{1, \ldots, Q\}$, $l \in \{1, \ldots, L\}$, where k is number of previous tokens to consider for attention, Q is the total number of attention heads per layer, and L is the total number of layers in the LLM.

Token probabilities (He et al., 2024; Shelmanov et al., 2025) is a vector of length m containing the log-probabilities of the m most likely tokens at generation step i:

$$F_{\text{tp}}(t_i) = \left\{ \log P(t \mid t_{<i}, \mathbf{x}) \right\} \tag{3}$$

where $\mathbf{x}$ is the sequence of input tokens, $t_{<i}$ is the sequence of tokens generated by the LLM up to token t_i, and t belongs to the set of m tokens with the highest probabilities according to the probability distribution P predicted by the LLM.

Token similarity (He et al., 2024) is a feature vector of token cosine similarities across all adjacent layer pairs

$$F_{\text{ts}}(t_i) = \{S^l(t_i)\}_{l=1}^{L-1} \tag{4}$$

where $S^l(t_i)$ is the cosine similarity between the unembedding-space embeddings of the top-m tokens from each adjacent pair of layers l and $l+1$

$$S^l(t_i) = \{\cos(E_{w_1}, E_{w_2})\} \tag{5}$$

where E_w is an embedding vector of token w from the unembedding matrix E, and w_1, w_2 are the tokens belonging to the set of m with the highest logit scores in the logit vectors of z_i^l of layer l and z_i^{l+1} of layer $l+1$ correspondingly. A logit vector $z_i^l = E(h_l(t_i))$ is obtained by applying the unembedding matrix E to the hidden state $h_l(t_i)$ of token t_i at layer l.

Output ranks (He et al., 2024) is a vector of the reciprocal of the rank $R^l(t_i)$ for each layer $l = 1, 2, \ldots, L$:

$$F_{\text{or}}(t_i) = \left\{ \left(R^l(t_i) \right)^{-1} \right\}_{l=1}^{L} \tag{6}$$

where $R^l(t_i)$ is the rank of token t_i in the descending list of logits $z_i^l = E(h_l(t_i))$ obtained by applying the unembedding matrix E to the hidden states $h_l(t_i)$ of token t_i at layer l.

[1] https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct

[2] https://huggingface.co/Qwen/QwQ-32B

[3] https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf

Estimator	Features	DSC 1.3B	DSC 6.7B	DSC 33B	CL 7B
Random	N/A	.313 ±.047	.201 ±.029	.205 ±.031	.303 ±.039
MSP	Token probabilities	.277 ±.031	.250 ±.028	.192 ±.022	.220 ±.026
Perplexity	Token probabilities	.240 ±.030	.215 ±.026	.163 ±.020	.202 ±.025
Attention Score	Attention weights	**.493** ±.054	**.336** ±.047	**.303** ±.045	**.454** ±.055
MTE	Entropy	.265 ±.031	.266 ±.038	.168 ±.020	.270 ±.030
MLP	Hidden states	.669 ±.071	.570 ±.069	.579 ±.089	.735 ±.056
Linear	Tok. sim., Out. ranks, Tok. prob., Hid. states	.677 ±.070	.550 ±.064	.471 ±.086	.742 ±.051
Linear	Lookback ratios	.520 ±.080	.385 ±.058	.552 ±.092	.621 ±.050
Transformer	Hidden states	.697 ±.065	.572 ±.067	**.590** ±.087	.756 ±.051
Transformer	Attention weights	.694 ±.091	.493 ±.083	.303 ±.038	.775 ±.054
Transformer	Hidden states, Attention weights	.651 ±.098	.594 ±.071	.555 ±.086	.798 ±.041
Transformer	Token probabilities, Attention weights	.630 ±.093	.535 ±.081	.426 ±.071	.743 ±.053
Transformer	Tok. prob., Att. weights, Hid. states	**.719** ±.085	.606 ±.068	.575 ±.085	**.811** ±.044
Transformer	Token probabilities	.292 ±.029	.213 ±.029	.244 ±.024	.270 ±.029
Transformer	Tok. sim., Out. ranks, Tok. prob., Hid. states	.673 ±.066	**.617** ±.058	.513 ±.071	.789 ±.046
Transformer	Lookback ratios	.665 ±.065	.551 ±.066	.546 ±.094	.761 ±.046

Table 2: PR-AUC measured on the mixed dataset across LLMs, estimators and features. The unsupervised and supervised UQ methods are separated with a horizontal line. "DSC" and "CL" abbreviations stand for DeepSeek-Coder and CodeLlama correspondingly. The highest values are highlighted in bold, the second-highest values are underlined. The values of standard deviation are collected using a statistical bootstrap.

Estimator	Train Data	Test Data	PR-AUC
Random	N/A	Synthesis	.176 ±.030
Transformer	Mixed	Synthesis	.344 ±.041
Transformer	Synthesis	Synthesis	**.349** ±.109
Transformer	Repair	Synthesis	.288 ±.056
Random	N/A	Repair	.255 ±.047
Transformer	Mixed	Repair	.561 ±.109
Transformer	Synthesis	Repair	.309 ±.066
Transformer	Repair	Repair	**.614** ±.094

Table 3: Transformer-based hallucination detector performance. The highest values are highlighted in bold, the second-highest values are underlined. The standard deviation is obtained throught statistical bootstrap.

Lookback ratios (Chuang et al., 2024) is a vector of values

$$F_{\text{lbl}}(t_i) = \left\{ LR^{q,l}(t_i) \right\} \tag{7}$$

for token t_i and all heads $q = 1, \ldots, Q$ and layers $l = 1, \ldots, L$ where $LR^{q,l}(t_i)$ is defined as

$$LR^{q,l}(t_i) = \frac{A^{q,l}_{\text{context}}(t_i)}{A^{q,l}_{\text{context}}(t_i) + A^{q,l}_{\text{gen}}(t_i)} \tag{8}$$

and $A^{q,l}_{\text{context}}(t_i)$ is the average attention to the context (prompt), $A^{q,l}_{\text{gen}}(t_i)$ is the average attention to previously generated tokens.

4.2 Models for Hallucination Detectors

Linear estimator is a logistic regression model f_{lr} that uses as input the feature vector F, obtained by averaging features over all tokens t_i in the generated code line S:

$$U_{\text{lr}}(S) = \text{f}_{\text{lr}} \left(\frac{1}{N} \sum_{i=1}^{N} F(t_i) \right) \tag{9}$$

Multi-Layer Perceptron (MLP) estimator. For token t_i of a line S of generated code, let $h(t_i)$ be the hidden state extracted from a specific layer of the LLM during generation. Let function f_{mlp} be the trained 3-layer perceptron that outputs a token uncertainty. The line uncertainty $U_{\text{mlp}}(S)$ is computed as

$$U_{\text{mlp}}(S) = \frac{1}{N} \sum_{i=1}^{N} f_{\text{mlp}} \left(h(t_j) \right) \tag{10}$$

where N is the length of the line in tokens. Notice that we experimented with training an MLP estimator on hidden states from multiple layers.

Transformer-based estimator. We propose an estimator that consists of a Transformer encoder Enc and a fully-connected classifier f_{cls} that outputs final logits:

$$U_{\text{te}}(S) = f_{\text{cls}} \left(\frac{1}{N} \sum_{i=1}^{N} \text{Enc}(\bar{F}_S) \right) \tag{11}$$

where $\bar{F}_S$ is the sequence of contextualized features

$$\bar{F}_S = \{ \tilde{f}_i + E_{\text{tr}} \}_{i=1}^{N} \tag{12}$$

and E_{tr} is a trainable embedding, $\tilde{f}_i$ are the projected features $F(t_i)$ using a fully-connected NN.

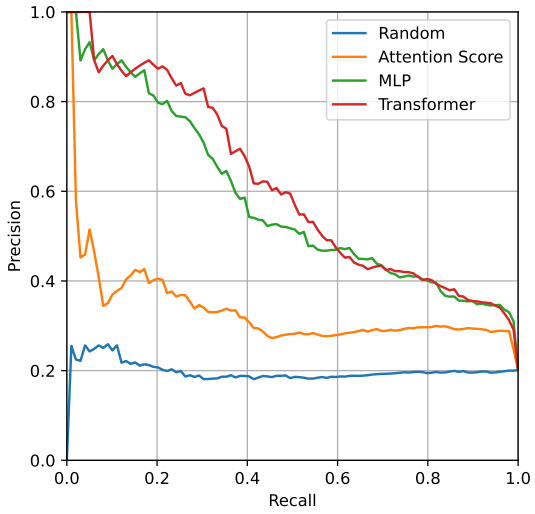

Figure 2: Precision-recall curves for Random, Attention score, MLP, and Transformer uncertainty estimations on DeepSeek-Coder 6.7B Instruct.

We used the framework of Shelmanov et al. (2025)[4] to implement the supervised estimators. Some of the existing hallucination detection methods (symbolic clustering (Sharma and David, 2025), execution-based verification (Tian et al., 2025), HalluCode (Liu et al., 2024), functional clustering (Ravuri and Amarasinghe, 2025), incoherence (Valentin et al., 2025)) for source code were not considered since they operate at the level of a programm. We did not consider the Collu-Bench (Jiang et al., 2024) uncertainty estimation method due to the absence of a publicly available implementation.

5 Experimental Setup

Large Language Models. The experiments were performed with instruct LLMs of two different families: DeepSeek-Coder[5] and CodeLlama[6], parameter numbers (1.3B, 6.7B, 33B variant of DeepSeek-Coder). Additionally, we performed the same experiments on Llama 3 8B Instruct[7] to compare the UQ quality of LLMs for coding and for general purposes.

[4] https://github.com/IINemo/llm-uncertainty-head

[5] https://huggingface.co/collections/deepseek-ai/deepseek-coder

[6] https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf

[7] https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct

Datasets. The experiments consider two domains of source code generation tasks: code synthesis in Python (MBPP, HumanEval) and code repair in Java (HumanEval Java, Defects4J). SWE-Bench subset of Collu-Bench was excluded from experiments due to excessively large length of its prompts that were causing out-of-memory (OOM) errors during experiments. The datasets that include two domains (mixed datasets) were collected for the instruct versions of DeepSeek-Coder (1.3B, 6.7B, 33B), CodeLlama 7B, and Llama 3 8B. For DeepSeek-Coder 6.7B, we collected separate datasets for each domain to analyze UQ quality on different domains. All the datasets were split into training (60%), validation (20%), and testing (20%) subsets. Table 1 provides detailed statistics on samples and labelled lines of code for the DeepSeek-Coder 6.7B datasets. The statistics of the datasets for other LLMs are provided in Table 7 of Appendix H.

To analyze UQ quality on various hallucination types, we considered the testing split of the mixed dataset of code generated by DeepSeek-Coder 6.7B with labelled hallucination types. The dataset happened to contain only examples that belong only to 3 out of 8 hallucination types: data compliance (60 examples), logic deviation (33 examples), and computational boundary (2 examples). Additionally, 8 examples have an undefined hallucination type since labelling LLMs achieved no consensus. Only logic deviation and data compliance hallucination types represent the majority in the testing dataset, so we considered only these two types.

Metrics. Hallucination detection is an imbalanced classification problem, as hallucinated lines are typically far less frequent than non-hallucinated ones (see Table 1). Due to this, we focused on PR-AUC (area under the precision-recall curve) as the primary quality metric to compare the UQ methods. To consider other aspects of the uncertainty quantification quality, we provide additional metric values of F1, Precision, Recall, Accuracy, Log Loss, ROC-AUC, and Prediction Rejection Area. To calculate the metrics, we used the LM-Polygraph library[8] (Fadeeva et al., 2023).

Hyperparameter selection for supervised estimators, Bayesian hyperparameter optimization was performed. We select hyperparameters across 25 different configurations. The details of the optimal

[8] https://github.com/IINemo/lm-polygraph

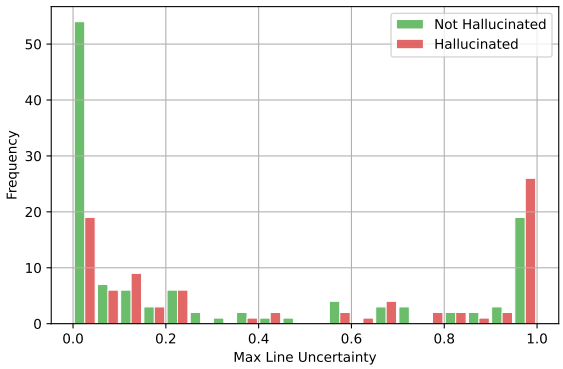

Figure 3: Histogram of maximum line uncertainties for hallucinated and correct functions.

hyperparameter configuration for the Transformer estimator are provided in Appendix C.

Basleines include the following unsupervised UQ methods: Maximum Sequence Probability (MSP) (Fadeeva et al., 2023), Perplexity (Kotti et al., 2025), Attention Score (Sriramanan et al., 2024), Maximum Token Entropy (MTE) (Fadeeva et al., 2023). See Appendix A for a detailed description.

6 Results and Discussion

6.1 Performance of Unsupervised Uncertainty Quantification Methods

Table 2 demonstrates PR-AUC values evaluated on the mixed dataset for different LLMs and estimators. The values of additional metrics are provided in Table 6 of Appendix G. The Attention Score method demonstrates the highest PR-AUC among all the unsupervised methods for all the considered LLMs. The other unsupervised methods fail to overcome the quality of a random estimation.

Most of the supervised methods demonstrate a higher quality of uncertainty quantification compared to the unsupervised methods. Among the supervised estimators, the Transformer estimator with hidden states, attention weights, and token probabilities features demonstrates the highest PR-AUC values for most of the LLMs. For DeepSeek-Coder 6.7B, the PR-AUC improvement of the Transformer estimator with hidden states, attention weights, and token probabilities features is 0.27 over the highest performing unsupervised method (Attention Score), and 0.036 over the second highest performing supervised method (MLP estimator with hidden states features). The precision-recall curves for the methods are compared in Figure 2.

6.2 Transformer Estimator Hyperparameters

Figure 9 and Figure 10 of Appendix F show the bar plots of Transformer estimator hyperparameters importance and correlation with PR-AUC correspondingly. The importance plot highlights the heads dimensionality as the most important architectural hyperparameter (11.7%). Most of the other architectural hyperparameters have an importance lower than 5%. The head dimensionality hyperparameter shows negative correlation (-0.388) with PR-AUC according to the correlation plot. However, most of the hyperparameters demonstrate weak correlation with PR-AUC.

6.3 Feature Importance

Given the values of PR-AUC, we performed the importance analysis of hidden states, attention weights, and token probability features. The results of ablation show very high relative feature importance of hidden states (0.572 PR-AUC), moderate importance of attention weights (0.493 PR-AUC), and low importance of token probabilities (0.213 PR-AUC).

6.4 Connection with Functional Correctness

In order to determine how estimated uncertainties correspond to the functional correctness of the generated code, we aggregated the line uncertainties to a single value. The uncertainties may be aggregated in two ways: by taking the mean or the maximum. According to our experiments, both aggregations make sense, but the maximum of uncertainties correlate stronger with functional incorrectness of generated code.

Figure 3 demonstrates the histogram of correct and hallucinated functions over maximum line uncertainties. In addition, Appendix E provides Figure 7 with the histogram over mean line uncertainties. Hallucinated functions show a much stronger presence at high maximum uncertainties close to 1. Thus, a high maximum uncertainty is a strong criterion of the source code correctness.

6.5 Effect of Various Factors on Uncertainty Quantification Quality

Number of Parameters. Table 2 shows PR-AUC decreasing as model size increases, which suggests larger models produce more subtle hallucinations.
LLM Family. The difficulty of hallucination detection may differ for the LLMs of different families. For example, our results show that CodeLlama

hallucinations are easier to detect than DeepSeek-Coder's (PR-AUC values are generally higher for CodeLlama in Table 2).

Domain (General-purpose). According to Table 4 in Appendix B, the Transformer estimator does not significantly differ from the supervised methods for general-purpose LLMs like Llama 3 8B (0.639 PR-AUC for Transformer estimator vs. 0.631 PR-AUC for MLP estimator).

Task Domain (Synthesis vs. Repair). The results provided in Table 3 show that the Transformer estimator trained on both domains generalizes well, while domain-specific training hurts cross-domain performance.

Hallucination Types. According to Figure 5 of Appendix D, logic deviation is easier to detect (PR-AUC 0.859) than data compliance (PR-AUC 0.652).

6.6 Error Analysis of an Uncertainty-based Code Classifier

To understand the performance of the UQ model, we use the value of uncertainty score (for both *mean* and *maximum* aggregation) in order to create two binary classifiers. A decision boundary and hence a threshold value for selecting erroneous examples was set to 0.5. Confusion matrix for *maximum* is shown in Figure 4. For *mean* aggregation, type 1 errors (when a hallucination in code is treated by the UQ model as a correct solution) are prevailing; also, there are very few predictions of hallucinations (see Figure 8 in Appendix E). For *maximum* aggrefation, one can observe a balance between type 1 and type 2 error rates in Figure 4 (46 false positive and 36 false negative cases). Therefore, below we focus on the analysis of errors for the *maximum* aggrefation. Additionally, we provide a histogram of lines of code uncertainties for hallucinated and correct lines in Figure 6 of Appendix E.

We analyze the top 46 false positives (type 1 errors) and the top 36 false negatives (type 2 errors) manually. False negatives (FN) are distributed equally among the tasks (17 code repair tasks and 19 code generation tasks). Interestingly, in all FN cases, the generated code matches the ground truth code exactly, while for the majority of false negatives (61%), the UQ model gives high scores (maximum uncertainty score above 0.9). Clearly, this issue can be resolved by testing the solution or using other external tools. So, real-world scenarios, when tests and specifications are available, type 2

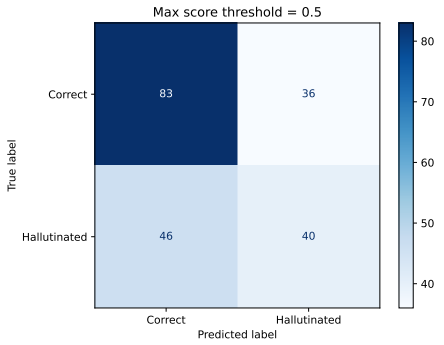

Figure 4: Confusion matrix of hallucinated function prediction via maximum uncertainty of the code lines.

errors should not be a problem. In contrast, false positives may lead to issues such as vulnerabilities in code. Unfortunately, our analysis of the false positive cases has not shown any patterns or interesting cases revealing why the UQ model returns low uncertainty score for wrong code. In most cases, the UQ model captures a general structure of the solution, but it fails to identify fine-grained hallucinations; that makes the wrong code look like correct one. The errors are related to slight logical deviations from a task specification.

7 Conclusion

We develop a supervised Transformer-based hallucination detector and demonstrate that it can effectively identify line-level hallucinations in source code, outperforming both existing unsupervised uncertainty quantification methods and prior supervised approaches. Our analysis shows that hidden states are the most informative features, and that maximum-based uncertainty aggregation provides a strong signal of functional incorrectness.

We release pre-trained hallucination detectors, an annotation pipeline for constructing training datasets and benchmarking hallucination detectors, and a novel source-code dataset annotated with line-level hallucinations to foster future research in this area.

Limitations

Due to the sensitivity of the diff utility to spelling, the proposed algorithm of code hallucination labelling for dataset composition tends to mark the lines of the hallucinated code as hallucinated, even though the line contains the correct variables, but the variables have names different from the names of the same variables in the ground-truth code. The supervised methods trained on the labelled dataset

with such an issue make the estimator mark certain correct lines of generated code as hallucinated.

Ethical Considerations

The proposed dataset of code hallucinations contains source code collected from publicly available sources. We release the proposed dataset and the source code for its collection under the MIT license.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Amos Azaria and Tom Mitchell. 2023. The internal state of an llm knows when it's lying. *arXiv preprint arXiv:2304.13734*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yung-Sung Chuang, Linlu Qiu, Cheng-Yu Hsieh, Ranjay Krishna, Yoon Kim, and James Glass. 2024. Lookback lens: Detecting and mitigating contextual hallucinations in large language models using only attention maps. *arXiv preprint arXiv:2407.07071*.

Ekaterina Fadeeva, Roman Vashurin, Akim Tsvigun, Artem Vazhentsev, Sergey Petrakov, Kirill Fedyanin, Daniil Vasilev, Elizaveta Goncharova, Alexander Panchenko, Maxim Panov, and 1 others. 2023. Lm-polygraph: Uncertainty estimation for language models. *arXiv preprint arXiv:2311.07383*.

Yarin Gal and 1 others. 2016. Uncertainty in deep learning.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Jinwen He, Yujia Gong, Zijin Lin, Cheng'an Wei, Yue Zhao, and Kai Chen. 2024. Llm factoscope: Uncovering llms' factual discernment through measuring inner states. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 10218–10230.

Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. 2024. Collu-bench: A benchmark for predicting language model hallucinations in code. *arXiv preprint arXiv:2410.09997*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Zoe Kotti, Konstantina Dritsa, Diomidis Spinellis, and Panos Louridas. 2025. The fools are certain; the wise are doubtful: Exploring llm confidence in code completion. *arXiv preprint arXiv:2508.16131*.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. 2023. Semantic uncertainty: Linguistic invariances for uncertainty estimation in natural language generation. In *The Eleventh International Conference on Learning Representations*.

Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Andrey Malinin and Mark J. F. Gales. 2021. Uncertainty estimation in autoregressive structured prediction. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Chaitanya Ravuri and Saman Amarasinghe. 2025. Eliminating hallucination-induced errors in llm code generation with functional clustering. *arXiv preprint arXiv:2506.11021*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Arindam Sharma and Cristina David. 2025. Assessing correctness in llm-based code generation via uncertainty estimation. *arXiv preprint arXiv:2502.11620*.

Artem Shelmanov, Ekaterina Fadeeva, Akim Tsvigun, Ivan Tsvigun, Zhuohan Xie, Igor Kiselev, Nico Daheim, Caiqi Zhang, Artem Vazhentsev, Mrinmaya Sachan, and 1 others. 2025. A head to predict and a

head to question: Pre-trained uncertainty quantification heads for hallucination detection in llm outputs. *arXiv preprint arXiv:2505.08200*.

André Silva and Fengjie Li. 2024. Humaneval-java: Transformed java defects dataset from humaneval. https://github.com/ASSERT-KTH/human-eval-java.

Oleg Somov and Elena Tutubalina. 2025. Confidence estimation for error detection in text-to-sql systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25137–25145.

Gaurang Sriramanan, Siddhant Bharti, Vinu Sankar Sadasivan, Shoumik Saha, Priyatham Kattakinda, and Soheil Feizi. 2024. Llm-check: Investigating detection of hallucinations in large language models. *Advances in Neural Information Processing Systems*, 37:34188–34216.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25300–25308.

Thomas Valentin, Ardi Madadi, Gaetano Sapia, and Marcel Böhme. 2025. Estimating correctness without oracles in llm-based code generation. *arXiv preprint arXiv:2507.00057*.

Roman Vashurin, Ekaterina Fadeeva, Artem Vazhentsev, Lyudmila Rvanova, Akim Tsvigun, Daniil Vasilev, Rui Xing, Abdelrahman Boda Sadallah, Kirill Grishchenkov, Sergey Petrakov, Alexander Panchenko, Timothy Baldwin, Preslav Nakov, Maxim Panov, and Artem Shelmanov. 2025. Benchmarking uncertainty quantification methods for large language models with lm-polygraph. *Transactions of the Association for Computational Linguistics*.

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

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503.

A Unsupervised UQ Methods

In order to provide mathematical formulations of the considered uncertainty quantification methods, we provide the following definitions. Let a sequence $S = (t_1, t_2, ..., t_N)$ of length Nbe a line of code consisting of tokens of the LLM-generated function. We also define LLM conditional probability for token t_i given the preceding tokens as $P(t_i|t_1, t_2, ..., t_{i-1})$.

Maximum Sequence Probability (MSP) (Fadeeva et al., 2023) estimates LLM uncertainty in a generated line of code as a negative sum of log probabilities for each token in the line:

$$U_{\text{msp}}(S) = -\sum_{i=1}^{N} \log P(t_i|t_1, t_2, ..., t_{i-1}) \quad (13)$$

Perplexity (Kotti et al., 2025) estimates LLM uncertainty as a negative sum of log probabilities normalized by the length of the line of code in tokens:

$$U_{\text{ppl}}(S) = -\frac{1}{N}\sum_{i=1}^{N} \log P(t_i|t_1, ..., t_{i-1}) \quad (14)$$

Attention Score (Sriramanan et al., 2024) calculates uncertainty using the following formula:

$$U_{\text{as}}(S) = -\frac{1}{H}\sum_{h=1}^{H} s_h \quad (15)$$

where H is the number of attention heads and s_h is the diagonal sum of log attention scores for the head h:

$$s_h = \sum_{i=1}^{T} \log\left(\left[A_h^l\right]_{ii}\right) \quad (16)$$

and A_h^l is the attention matrix for head h at layer l.

Maximum Token Entropy (MTE) (Somov and Tutubalina, 2025) defines LLM uncertainty as the maximum token entropy within a line:

$$U_{\text{mte}}(S) = \max_{t \in S} H_t \quad (17)$$

where H_t is a token entropy calculated as

$$H_t = -\sum_{v \in V} p_{t,v} \cdot \log p_{t,v} \quad (18)$$

and $p_{t,v} = p(v \mid y_{<t}, x)$ is the probability of token v at position t, V is the model vocabulary, x and y are input and output tokens correspondingly.

B Code Hallucination Detection for General-purpose LLMs

Estimator	Features	PR-AUC
Attention Score	AW	.369 $\pm$.057
MLP	HS	.638 $\pm$.058
Transformer	TP, AW, HS	.630 $\pm$.049

Table 4: PR-AUC measured on the mixed dataset for general-purpose Llama 3 8B Instruct. "AW" stands for attention weights, "HS" for hidden states, and "TP" for token probabilities. The values of standard deviation are collected using a statistical bootstrap.

C Optimal Hyperparameters of the Transformer Estimator

Hyperparameter	Value
Learning Rate	5.0e-05
Positive Weight	16
Weight Decay	0
Max Gradient Norm	2
Epochs	4
Warmup Ratio	0.1
Gradient Accumulation Steps	2
Head Dimensionality	2048
Number of Heads	16
Top-N Probabilities	1
Pooling	Enabled
Attention History Size	8
Dropout	0.05
Number of Layers	2
Hidden States Source Layers	32, 28, 23
Attention Weights Source Layers	32, 28

Table 5: Optimal hyperparameters for the Transformer estimator with hidden states, attention weights, and token probability features. Training (top) and architectural (bottom) hyperparameters are separated with a horizontal line.

D Transformer Estimator Performance on Various Hallucination Types

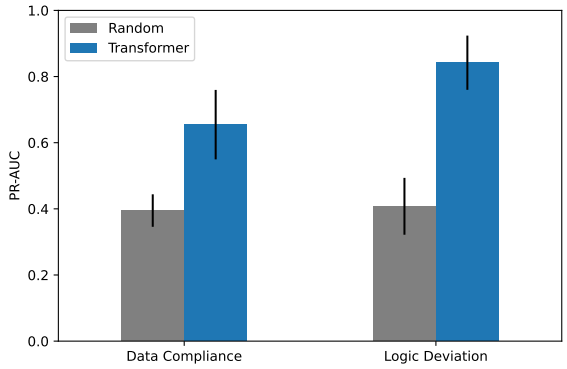

Figure 5: Transformer estimator performance across hallucination types. The values of standard deviation are collected using a statistical bootstrap.

E Error Analysis of Transformer Estimator

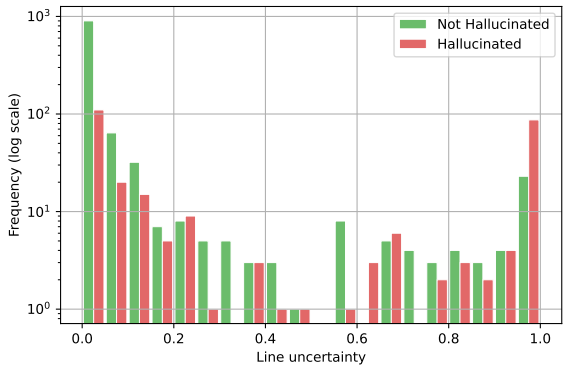

Figure 6: Histogram of line uncertainties for hallucinated and correct lines of code.

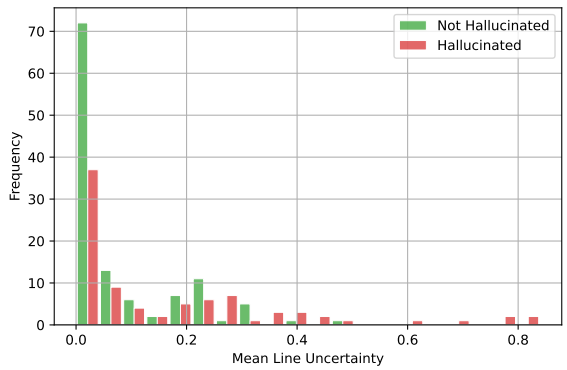

Figure 7: Histogram of mean line uncertainties for hallucinated and correct functions.

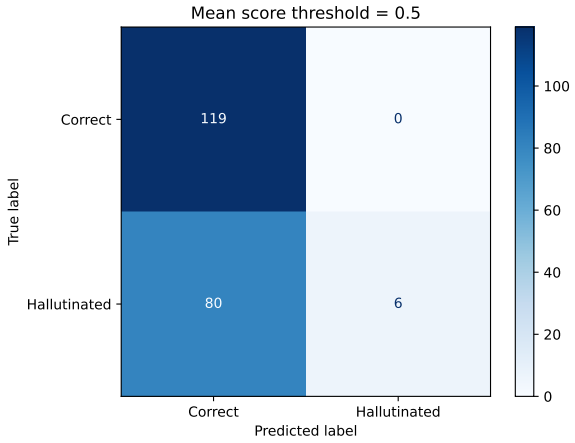

Figure 8: Confusion matrix of hallucinated function prediction using mean uncertainty of the lines of code.

F Effect of Transformer Estimator Hyperparameters on PR-AUC metric

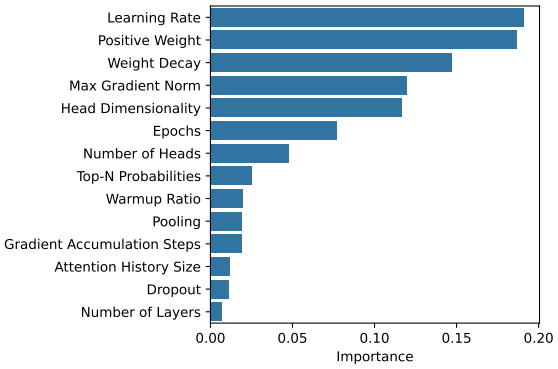

Figure 9: Transformer estimator hyperparameters importance relative to PR-AUC metric

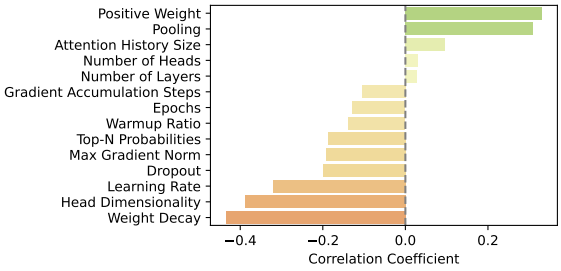

Figure 10: Transformer estimator hyperparameters correlation with PR-AUC metric

G Additional Metric Values of Uncertainty Quantification Quality

Estimator	Features	F1	Prec.	Rec.	Acc.	Log Loss	ROC AUC	PRA	PRA 50%
Random	N/A	0.481	0.485	0.477	0.489	1.029	0.474	0.225	0.205
MSP	TP	0.485	0.474	0.496	0.782	0.945	0.581	0.193	0.174
Perplexity	TP	0.441	0.398	0.494	0.790	0.979	0.520	0.219	0.194
Attention Score	AW	0.644	0.900	0.502	0.799	0.945	0.700	0.097	0.164
MTE	Entropy	0.550	0.585	0.520	0.788	0.640	0.551	0.203	0.187
MLP	HS	0.703	0.681	0.726	0.777	0.454	0.836	0.060	0.113
Linear	TS, TP, OR, HS	0.662	0.667	0.658	0.788	0.514	0.827	0.062	0.116
Linear	Lookback Ratios	0.613	0.591	0.636	0.648	0.537	0.719	0.092	0.154
Transformer	HS	0.718	0.748	0.690	0.835	0.855	0.844	0.060	0.107
Transformer	AW	0.665	0.631	0.703	0.642	0.590	0.790	0.073	0.129
Transformer	HS, AW	0.705	0.749	0.666	0.832	0.763	0.845	0.058	0.108
Transformer	TP, AW	0.656	0.660	0.652	0.784	0.413	0.809	0.068	0.121
Transformer	TP, AW, HS	0.716	0.764	0.673	0.838	0.634	0.843	0.059	0.109
Transformer	TP	0.168	0.101	0.500	0.201	0.762	0.572	0.177	0.183
Transformer	TS, TP, OR, HS	0.712	0.677	0.750	0.751	0.541	0.859	0.054	0.105
Transformer	Lookback Ratios	0.678	0.643	0.718	0.680	0.705	0.821	0.064	0.118

Table 6: Performance comparison of various estimators with different features on the mixed dataset for DeepSeek-Coder 6.7B Instruct. "PRA" metric stands for prediction rejection area. "TP" feature stands for token probabilities, "AW" for attention weights, "HS" for hidden states, "TS" for token similarities, and "OR" for output ranks. Metrics are calculated at a threshold of 0.5.

H Statistics of the Datasets for Other LLMs

Dataset	Split	Total Samples	Total Lines	Hallucinated Lines	Correct Lines
DSC 1.3B, Syn & Repair	Train	807	5591	1485 (27%)	4106 (73%)
	Validation	202	1469	370 (25%)	1099 (75%)
	Test	253	1799	545 (30%)	1254 (70%)
	Total	1262	8859	2400 (27%)	6459 (73%)
CL 7B, Syn & Repair	Train	803	5793	1363 (24%)	4430 (76%)
	Validation	201	1377	258 (19%)	1119 (81%)
	Test	252	1827	518 (28%)	1309 (72%)
	Total	1256	8997	2139 (24%)	6858 (76%)
Llama 3 8B, Syn & Repair	Train	766	4910	979 (20%)	3931 (80%)
	Validation	192	1197	192 (16%)	1005 (84%)
	Test	240	1630	305 (19%)	1325 (81%)
	Total	1198	7737	1476 (19%)	6261 (81%)
DSC 33B, Syn & Repair	Train	572	3922	937 (24%)	2985 (76%)
	Validation	143	938	153 (16%)	785 (84%)
	Test	179	1379	319 (23%)	1060 (77%)
	Total	894	6239	1409 (23%)	4830 (77%)

Table 7: Statistics of the collected datasets of line-level code hallucinations of the instruct versions of DeepSeek-Coder, CodeLlama, and Llama 3 LLMs. The dataset consists of subsets specific to the code generation domains; the subset names are provided in the corresponding column. The *Hallucinated Lines* and *Correct Lines* columns providepercentage information relative to the *Total Lines* column in the parentheses.

I Code Generation Prompt Examples from the Proposed Dataset

```
<|begin_of_sentence|>You are an AI programming assistant, utilizing the Deepseek Coder model, developed
by Deepseek Company, and you only answer questions related to computer science. For politically
sensitive questions, security and privacy issues, and other non-computer science questions, you will
refuse to answer
### Instruction:
Write a function to count number of unique lists within a list.

Respond only with code. Start the response with "```python
def unique_sublists(list1):" and complete it.

### Response:
```

Figure 11: Example of the code synthesis prompt for DeepSeek-Coder. The prompt is inspired by the prompts proposed in Collu-Bench (Jiang et al., 2024).

````
<|begin_of_sentence|>You are an AI programming assistant, utilizing the Deepseek Coder model, developed
by Deepseek Company, and you only answer questions related to computer science. For politically
sensitive questions, security and privacy issues, and other non-computer science questions, you will
refuse to answer
### Instruction:
You will be provided with a PROBLEM DESCRIPTION, the FUNCTION that is intended to solve the problem yet
contains a bug, with the BUGGY CODE highlighted between <bug> and </bug> tags. Your task is to analyze
the entire FUNCTION and the FUNCTION, then generate the FIXED BUGGY CODE.

The generated FIXED BUGGY CODE will directly replace the BUGGY CODE within the FUNCTION. Please ensure
that the syntax is correct and that no additional code is produced beyond the FIXED BUGGY CODE, as this
could lead to syntax errors when the FIXED BUGGY CODE is inserted back into the FUNCTION.

Respond only with code. Start the response only with "```java" and complete it.

PROBLEM DESCRIPTION
You are given a list of integers.
Write a function next_smallest() that returns the 2nd smallest element of the list.
Return null if there is no such element.

Examples:
next_smallest({1, 2, 3, 4, 5}) returns 2
next_smallest({5, 1, 4, 3, 2}) returns 2
next_smallest({}) returns null
next_smallest({1, 1}) returns null


FUNCTION
```java
public class NEXT_SMALLEST {
    public static Integer next_smallest(int[] lst) {

<bug>
        List<Integer> numbers = new ArrayList<Integer>();
</bug>
        for (Integer n : lst)
            numbers.add(n);
        Integer[] no_duplicate = numbers.toArray(new Integer[] {});
        Arrays.sort(no_duplicate);

        if (no_duplicate.length < 2)
            return null;
        return no_duplicate[1];
    }
}


BUGGY CODE
```java
        List<Integer> numbers = new ArrayList<Integer>();
```

FIXED BUGGY CODE

### Response:
````

Figure 12: Example of the code repair prompt for DeepSeek-Coder. The prompt is inspired by the prompts proposed in Collu-Bench (Jiang et al., 2024).
````

J Prompt for Labelling Hallucination Types

````
You're an expert in hallucinations classification in code.

You're given the following code with some hallucinations: ```python
{model_output}
```

The execution feedback is: {feedback}
The ground truth code is: ```python
{closest_gt}
```

Classify the hallucination into one of the classes:
1. Data compliance hallucination. Definition: vague understanding of the data types and parameter values
of the objects being manipulated, resulting in generated code that attempts to perform type-mismatched
or rule-violating operations.

2. Structure access hallucination. Definition: misinterpret the data structures of the objects being
manipulated, leading to generated code that attempts to access non-existent array indices or dictionary
keys.

3. External source hallucination. Definition: Memory-related issues with external knowledge sources,
resulting in generated code that attempts to import non-existent modules or fails to correctly load
modules from other paths.

4. Identity hallucination. Definition: Biased memories or lack sufficient understanding of the context,
leading to generated code that references undefined variables, accesses non-existent object properties,
or uses unassigned variables in local scopes.

5. Computational boundary hallucination. Definition: blur recognition of numerical calculation limits
and iteration endpoints.

6. Physical constraint hallucination. Definition: underestimate resource consumption during data
processing operations, causing code failure due to exceeding memory capacity, stack depth, or other
physical constraints.

7. Logic deviation hallucination. Definition: generation of code that lacks sufficient logical
consideration or contradicts the intended instructions.

8. Logic breakdown hallucination. Definition: struggle to interpret or maintain a continuous
understanding of context during code generation. This indicates that the models may lose direction while
generating code, making it difficult to maintain strict consistency of contextual information

Return only the class name from the provided list. Do not include definition
````

Figure 13: Prompt template for hallucination type. The list of hallucination types refers to CodeHalu (Tian et al., 2025).

K Code Repair Hallucination Example

TASK:

Create a function that returns true if the last character of a given string is an alphabetical character and is not a part of a word, and false otherwise.

FUNCTION TO FIX:

```
1:  public class CHECK_IF_LAST_CHAR_IS_A_LETTER {
2:      public static boolean check_if_last_char_is_a_letter(String txt) {
3:          String[] parts = txt.split(" ");
4:          String last = " ";
<bug>
5:          last = parts[parts.length - 1];
6:          if (txt.charAt(txt.length() - 1) == ' ')
7:              last = " ";
8:          if (txt.length() == 1)
9:              last = " ";
10:         int last_char_pos = Character.toUpperCase(last.charAt(0)) - 'a';
11:         return (last.length() == 1) || (0 <= last_char_pos || last_char_pos <= 25);
</bug>
12:     }
13: }
```

CORRECTLY FIXED FUNCTION

```
1:  public class CHECK_IF_LAST_CHAR_IS_A_LETTER {
2:      public static boolean check_if_last_char_is_a_letter(String txt) {
3:          String[] parts = txt.split(" ");
4:          String last = " ";
<bug>
5:          if (parts.length != 0)
6:              last = parts[parts.length - 1];
7:          if (txt.length() != 0 && txt.charAt(txt.length() - 1) == ' ')
8:              last = " ";
9:          if (txt.length() == 0)
10:             last = " ";
11:         int last_char_pos = Character.toLowerCase(last.charAt(0)) - 'a';
12:         return (last.length() == 1) && (0 <= last_char_pos && last_char_pos <= 25);
</bug>
13:     }
14: }
```

INCORRECTLY FIXED FUNCTION

```
1:  public class CHECK_IF_LAST_CHAR_IS_A_LETTER {
2:      public static boolean check_if_last_char_is_a_letter(String txt) {
3:          String[] parts = txt.split(" ");
4:          String last = " ";
<bug>
5:          last = parts[parts.length - 1];
6:          if (txt.length() == 1)
7:              last = " ";
8:          int last_char_pos = Character.toUpperCase(last.charAt(0)) - 'a';
9:          return (last.length() == 1) || (0 <= last_char_pos || last_char_pos <= 25);
</bug>
10:     }
11: }
```

Figure 14: An example of hallucination detection for code repair tasks (HumanEval Java). Correct lines of the hallucinated solution are highlighted in green, while hallucinated lines are highlighted in red. Line 5 of the incorrectly fixed function is hallucinated because there should be an if condition before it. Line 6 of the incorrectly fixed function is hallucinated because there is no such condition in the correctly fixed function. Line 8 of the incorrectly fixed function is hallucinated because it should have used the toLowerCase function. Line 9 of the incorrectly fixed function is hallucinated because it should contain a conjunction (&&) in the condition instead of a disjunction (||).