Efficient Hallucination Detection in Automatic Code Generation

Georgii Andriushchenko[1] **Roman Garaev**[1] **Lyudmila Rvanova**[2,3]
Artem Shelmanov[4 *] **Vladimir Ivanov**[1 *]
[1]Research Center of the AI Institute, Innopolis University
[2]Laboratory for Analysis and Controllable Text Generation Technologies, RAS
[3]FusionBrain Lab, AXXX
[4]MBZUAI

Abstract

Large language models (LLMs) frequently produce source code that seems correct and well-formed, yet includes hallucinated elements that cause downstream test failures. In this study, we benchmark state-of-the-art uncertainty quantification methods and existing baselines for the task of hallucination detection in source code and introduce a diff-based pipeline to construct a code dataset annotated with line-level hallucinations. Building on this, we train a lightweight Transformer-based detector that uses LLM internal representations to identify hallucinations, substantially outperforming existing methods across several code generation domains. The detector also shows particular promise for enabling self-correction in LLM-based coding agents. We release the first publicly available dataset of line-level code hallucinations, along with the corresponding source code and trained hallucination detectors https://github.com/datapaf/CodeHallucinationDetection

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). In this study, we follow the definition of a hallucination given by Jiang et al. (2024) as *"a phenomenon where an LLM generates code that is plausible and fluent but is actually incorrect, unfaithful, or fabricated"*. Uncertainty quantification (UQ) methods offer a promising way to identify unreliable and potentially hallucinated model outputs (Gal, 2016; Malinin and Gales, 2021). Although hallucination detection has been studied extensively in natural language generation tasks (Kuhn et al., 2023; Fadeeva et al., 2023; Vashurin et al., 2025), it remains relatively underexplored in source code generation, especially at fine-grained levels, such as individual lines of code.

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 the source code generated by DeepSeek-Coder 1.3B Instruct for the code synthesis task. The example is taken from the HumanEval subset of Collu-Bench (Jiang et al., 2024). 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.

Correspondence to: georgyandryuschenko@gmail.com
*Equal contribution

Findings of the Association for Computational Linguistics: ACL 2026, pages 43197–43220
July 2-7, 2026

In this work, we conduct a comprehensive evaluation of UQ methods and other baselines 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. Using this pipeline, we build a large-scale annotated dataset and train a lightweight Transformer-based hallucination detector that leverages LLM inner representations as input features. 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 the evaluation of hallucination detection methods while using arbitrary LLMs as the code generators.

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 and code generation domains (code synthesis and code repair). The empirical investigation also includes an analysis of the most impactful features for detecting hallucinations in code.

2 Related Work

Existing detectors of hallucinations in source code predominantly operate at the code snippet level and determine whether an entire generated program is hallucinated (Sharma and David, 2025; Ravuri and Amarasinghe, 2025; Tian et al., 2025; Valentin et al., 2025). They also often rely on execution-based signals such as runtime errors or test outcomes. While these methods can be effective for coarse-grained validation, they provide little insight into where hallucinations occur within the code and are inherently constrained by the availability and coverage of executable tests. To the best of our knowledge, Collu-Bench (Jiang et al., 2024) is the only work that attempts to localize hallucinations at a finer granularity. Nevertheless, it identifies only the starting point of a hallucination and labels all subsequent code as hallucinated, which may be insufficiently informative for debugging.

UQ has been widely applied for hallucination detection in natural language generation (Kuhn et al., 2023; Fadeeva et al., 2023; Vashurin et al., 2025). Existing UQ methods include unsupervised techniques (Fadeeva et al., 2023; Kotti et al., 2025; Sriramanan et al., 2024; Somov and Tutubalina, 2025) and supervised models trained on internal LLM representations (Azaria and Mitchell, 2023; Chuang et al., 2024; He et al., 2024; Vazhentsev et al., 2025b,a; Shelmanov et al., 2025). However, UQ methods remain largely underexplored in the source code domain, particularly for fine-grained hallucination localization. This work fills this gap by proposing a supervised approach to line-level hallucination in LLM-generated source code.

3 Dataset Construction

3.1 Base Dataset

We use 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 et al., 2021), HumanEval Java (Silva and Li, 2024), Defects4J (Just et al., 2014), and SWE-bench (Jimenez et al., 2024). The code was generated by DeepSeek-Coder (Guo et al., 2024), CodeLlama (Rozière et al., 2024), 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 K. We chose this dataset because 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.

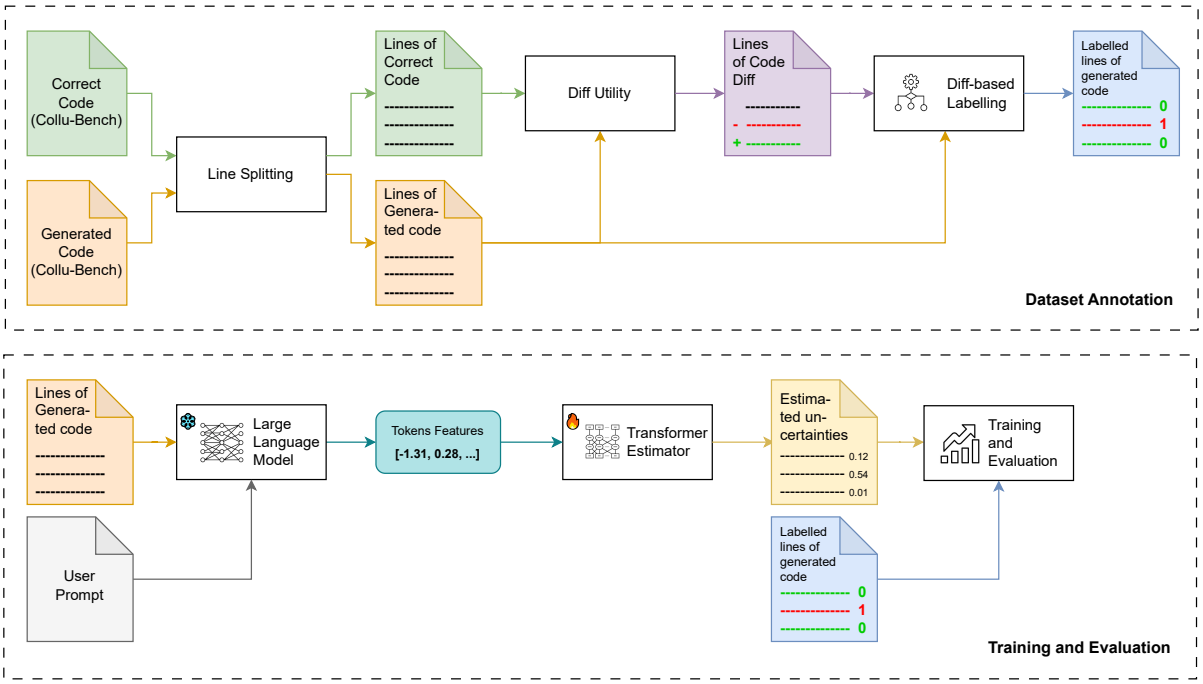

Figure 2: Overview of the dataset annotation, training, and evaluation. *Correct code* and *generated code* are obtained from Collu-bench (Jiang et al., 2024) for particular LLM. White rectangles represent the stages of the pipeline, and intermediate artifacts between the stages are colored in blue. During the training, only the weights of the Transformer estimator are updated, while the weights of the LLM are frozen. Training and evaluation are described in the single pipeline since evaluation follows a process similar to training.

3.3 Pipeline for Hallucination Annotation

A key challenge in constructing a dataset of line-level hallucinations is reliable annotation. We propose an automatic labeling procedure that identifies line hallucinations using the `diff` utility according to the following algorithm:

1. Split generated and ground-truth code snippets into lines.

2. Compute the `diff` between the lines of the generated and ground-truth code snippets.

3. Label all the lines that should be removed from the generated code snippet (marked as "`-`" in the diff) as hallucinating.

4. Label all the first lines that come immediately after line additions (marked as "`+`" in the diff) as hallucinations. If there are line removals right before line additions, label the first line as correct.

Following the proposed algorithm, we annotated code generated by DeepSeek-Coder, CodeLlama, and Llama models, resulting in a publicly available dataset of LLM-generated code with line-level hallucination labels. Figure 2 illustrates the hallucination annotation pipeline. Examples of line-level code hallucination labels are provided in Figure 1 for the code generation task and in Figure 14 of Appendix L for the code repair task. The statistics of the annotated datasets are provided in Table 1 and Table 12 of Appendix J.

The proposed diff-based algorithm for dataset annotation is cheaper and more effective compared to the annotation obtained through LLM-as-a-judge. The main limitation of the proposed diff-based annotation algorithm is its sensitivity to syntactical differences in the generated and ground-truth source code. This limitation could be partially mitigated through source-code normalization. Since Collu-Bench already provides normalized code, we do not apply additional normalization in our experiments. The quality of diff-based annotation with respect to manual annotation is discussed in Appendix C.

4 Line-level Code Hallucination Detection

4.1 Method

Given the input prompt $\mathbf{x}$ and the LLM-generated source code $\mathbf{y}$ that contains a line of code $s \in \mathbf{y}$, we consider uncertainty for a line of code as a function

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

$U(s) \in [0, 1]$ where larger values of U indicate that s is more likely to be hallucinated.

For detecting line-level hallucinations in LLM-generated code, we adapt Uncertainty Head from (Shelmanov et al., 2025). It is a supervised hallucination detector based on the Transformer architecture and the internal states of LLMs as features. Since it uses only the capabilities of the LLM itself and approximates the estimate of errors made by an LLM, we can consider it a form of supervised confidence estimator. The detailed formal description of the Uncertainty Head and our method is provided in Appendix B.2. The choice of this method is motivated by the fact that it has previously demonstrated a high quality of hallucination detection in LLM-generated natural language texts.

4.2 Input Features

To detect hallucinations, we considered several types of features used in the previous studies of LLM hallucination detection: *hidden states* (Azaria and Mitchell, 2023; He et al., 2024; Shelmanov et al., 2025), *attention weights* (Shelmanov et al., 2025), *token probabilities* (He et al., 2024; Shelmanov et al., 2025), *token similarities* (He et al., 2024), *output ranks* (He et al., 2024), *and lookback ratios* (Chuang et al., 2024). The detailed formal description of the mentioned features is provided in Appendix B.1. To achieve the highest quality of hallucination detection, we considered various combinations of the features. See Section 5.4 for the details of the feature combinations.

4.3 Detector Architecture

The detection using the Transformer estimator is organized according to Shelmanov et al. (2025). First, a feature vector $F(t)$ is produced for each token t in the generated source code $\mathbf{y}$. The feature vector $F(t)$ is then projected to vector $\tilde{f}_t$ using a fully-connected layer FC_{proj} to be able to pass it to the Transformer encoder:

$$\tilde{f}_t = FC_{\text{proj}}(F(t)) \tag{1}$$

Then, each projected feature vector $\tilde{f}_t$ for all $t \in s$ is added to the trainable line position embeddings E_s specific for each line to obtain a contextualized feature vector $\tilde{f}_{t,s}$:

$$\tilde{f}_{t,s} = \tilde{f}_t + E_s \tag{2}$$

Next, the feature vector $\tilde{f}_{t,s}$ is passed to Transformer (Vaswani et al., 2017) encoder T and averaged to obtain contextualized vector representation h_s of token $t \in s$:

$$h_s = \frac{1}{|s|} \sum_{t \in s} T(\{\tilde{f}_{t,s}\}_{t \in \mathbf{x} \circ \mathbf{y}}) \tag{3}$$

The uncertainty for a line of code s is estimated using a multi-layer perceptron MLP_{clf} and sigmoid activation function σ:

$$U(s) = \sigma\left(\text{MLP}_{\text{clf}}(h_s)\right) \tag{4}$$

To prevent overfitting, the dropout (Srivastava et al., 2014) regularization for MLP_{clf} is provided.

Table 8 in Appendix E demonstrates the optimal architectural hyperparameters (e.g., dimensionality) of the Transformer-based hallucination detector that were found using hyperparameter optimization (see Section 5 for optimization details).

4.4 Transformer Estimator Training

To train the Transformer-based hallucination detector on the input features, we used the framework of Shelmanov et al. (2025)[1]. The estimator was trained by minimizing *binary cross-entropy loss* using the Adam optimizer with weight decay, a linear learning rate schedule, and warmup. The training parameters, such as the number of epochs, learning rate, weight decay, warmup ratio, weight of the positive class, maximum gradient norm, and gradient accumulation steps, were subject to hyperparameter optimization. Figure 2 presents the overall pipeline.

5 Experimental Setup

5.1 Large Language Models

The main experiments were conducted with open-source instruction-tuned LLMs from two different families: CodeLlama 7B[2] and DeepSeek-Coder[3] 1.3B, 6.7B, and 33B. Additionally, we conducted experiments with Llama-3-8B-Instruct[4] to compare hallucination-detection methods across both general-purpose and code-focused LLMs.

5.2 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). The SWE-Bench subset of Collu-Bench was excluded from the experiments due to the excessively large length of its prompts, which caused out-of-memory (OOM) errors during the 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 across different domains. All the datasets were split into training (60%), validation (20%), and testing (20%) subsets. Table 1 provides detailed statistics on samples and labeled lines of code for the DeepSeek-Coder 6.7B datasets. The statistics of the datasets for other LLMs are provided in Table 12 of Appendix J.

5.3 Baselines

We compare the performance of our Transformer-based supervised hallucination detector with existing *unsupervised*, *supervised*, and *other* baselines. The unsupervised group includes popular UQ methods such as the maximum sequence probability (MSP) (Fadeeva et al., 2023), perplexity (Kotti et al., 2025), Attention Score (Sriramanan et al., 2024), and maximum token entropy (MTE) (Somov and Tutubalina, 2025). Appendix A provides details of the unsupervised methods. The considered supervised baselines include a linear classifier on the set of token similarities, output ranks, token probabilities, and hidden states features (He et al., 2024), Lookback lens – a linear estimator on top of the lookback ratios features (Chuang et al., 2024), and SAPLMA (Azaria and Mitchell, 2023) – a multi-layer perceptron (MLP) estimator that uses hidden states as features. Appendix B.3 contains details of the mentioned supervised baselines. The group of other baselines includes previously applied approaches to code hallucination detection (Agarwal et al., 2025), such as the CodeBERT (Feng et al., 2020) classifier using code tokens as features and basic prompting methods (see Appendix M for details). Additionally, we considered a random estimator selecting uncertainty from the Uniform(0, 1) distribution with a fixed seed.

5.4 Transformer-Based Detector Features

The combination of token probabilities (TP), hidden states (HS), and attention weights (AW) has previously been shown to be effective for hallucination detection (Shelmanov et al., 2025). We therefore adopted this feature set for our Transformer-based hallucination detector. To assess the contribution of different features, we compared it against other previously proposed combinations: (1) token similarity, output ranks, token probabilities, and hidden states (He et al., 2024); and (2) lookback ratios (Chuang et al., 2024). In addition, we evaluated all subsets of the feature set TP, HS, AW to analyze the contribution of each individual feature.

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

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

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

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

Group	Estimator	Features	DSC 1.3B	DSC 6.7B	DSC 33B	CL 7B	Average
Other baselines	Random	N/A	.313 ±.047	.201 ±.029	.205 ±.031	.303 ±.039	.256 ±.037
	Prompting	Zero-shot prompt	.450 ±.018	.273 ±.028	.166 ±.016	.263 ±.028	.288 ±.023
	Prompting	One-shot prompt	.462 ±.026	.263 ±.029	.254 ±.033	.269 ±.026	.312 ±.029
	Prompting	CoT prompt	.417 ±.031	.260 ±.051	.047 ±.013	.231 ±.082	.239 ±.045
	CodeBERT	Tokens	.539 ±.038	.359 ±.034	.339 ±.037	.471 ±.040	.472 ±.037
Unsupervised baselines	MSP	TP	.277 ±.031	.250 ±.028	.192 ±.022	.220 ±.026	.235 ±.027
	Perplexity	TP	.240 ±.030	.215 ±.026	.163 ±.020	.202 ±.025	.205 ±.025
	Attention Score	AW	.493 ±.054	.336 ±.047	.303 ±.045	.454 ±.055	.397 ±.050
	MTE	Entropy	.265 ±.031	.266 ±.038	.168 ±.020	.270 ±.030	.242 ±.030
Supervised baselines	Linear	TS, OR, TP, HS	.677 ±.070	.550 ±.064	.471 ±.086	.742 ±.051	.610 ±.068
	Linear	LR	.520 ±.080	.385 ±.058	.552 ±.092	.621 ±.050	.520 ±.070
	MLP	HS	.669 ±.071	.570 ±.069	.579 ±.089	.735 ±.056	.638 ±.056
Our detectors	Transformer	HS	.697 ±.065	.572 ±.067	**.590** ±.087	.756 ±.051	.654 ±.068
	Transformer	AW	.694 ±.091	.493 ±.083	.303 ±.038	.775 ±.054	.566 ±.067
	Transformer	TP	.292 ±.029	.213 ±.029	.244 ±.024	.270 ±.029	.255 ±.028
	Transformer	HS, AW	.651 ±.098	.594 ±.071	.555 ±.086	.798 ±.041	.650 ±.074
	Transformer	HS, TP	.684 ±.067	.555 ±.071	.560 ±.085	.801 ±.044	.650 ±.067
	Transformer	TP, AW	.630 ±.093	.535 ±.081	.426 ±.071	.743 ±.053	.584 ±.075
	Transformer	TP, AW, HS	**.719** ±.085	.606 ±.068	.575 ±.085	**.811** ±.044	**.678** ±.071
	Transformer	TS, OR, TP, HS	.673 ±.066	**.617** ±.058	.513 ±.071	.789 ±.046	.648 ±.060
	Transformer	LR	.665 ±.065	.551 ±.066	.546 ±.094	.761 ±.046	.631 ±.068

Table 2: PR-AUC measured on the mixed dataset across LLMs, estimators, and features. The first column indicates the groups of detectors that are separated by a horizontal line. “DSC" and “CL" abbreviations in the column names stand for DeepSeek-Coder and CodeLlama respectively. The features, such as token probabilities (TP), attention weights (AW), hidden states (HS), token similarities (TS), output ranks (OR), and lookback ratios (LR), are abbreviated. In each column, the highest values are highlighted in bold, while the second-highest values are underlined. The values of standard deviation are collected using a statistical bootstrap.

5.5 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 hallucination detection methods. To consider other aspects of the uncertainty quantification quality, we provide the results with some additional metrics in Appendix I. The evaluation was done using the LM-Polygraph framework[5] (Fadeeva et al., 2023; Vashurin et al., 2025).

5.6 Hyperparameter Selection for Supervised Hallucination Detectors

A Bayesian hyperparameter optimization (Snoek et al., 2012) was performed to find the optimal hyperparameters for all the considered supervised estimators. We select hyperparameters across 25 different configurations. The details of the optimal hyperparameter configuration for the Transformer estimator are provided in Appendix E.

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

6 Results and Discussion

6.1 Baselines Performance

Table 2 demonstrates PR-AUC values obtained on the mixed dataset for various LLMs and hallucination detectors. The values of additional metrics are provided in Table 11 of Appendix I. The Attention Score method demonstrates the highest PR-AUC among all the unsupervised methods for all the considered LLMs. Other unsupervised methods fail to surpass the quality of random estimations. Most of the supervised methods demonstrate a higher quality of hallucination detection compared to the unsupervised methods. Among the supervised estimators, the *Transformer-based estimator with hidden states, attention weights, and token probabilities features demonstrates the highest PR-AUC values for most LLMs*. For DeepSeek-Coder 6.7B, the PR-AUC improvement of the Transformer-based detector 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 CodeBERT baseline achieves an average PR-AUC of 0.472, which is between the

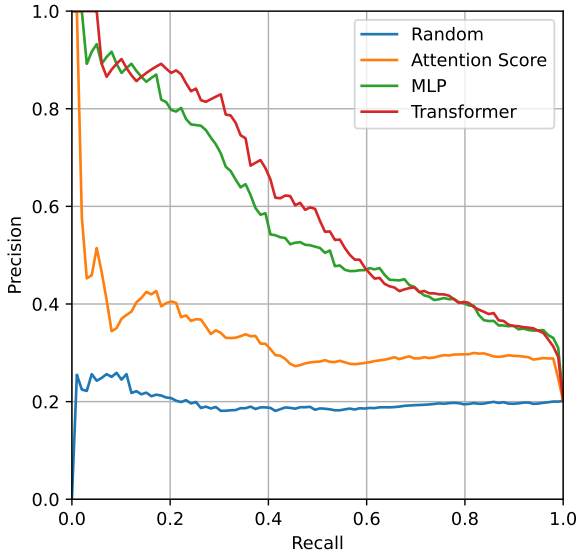

Figure 3: Precision-recall curves for Random, Attention score, MLP, and Transformer uncertainty estimators on DeepSeek-Coder 6.7B Instruct.

unsupervised and other supervised baselines. The precision-recall curves for the methods are compared in Figure 3.

Prompting methods appear to be unstable and demonstrate low performance. In unstable cases, the LLM may provide output in an unexpected format that is hard to parse or may simply refuse to answer. This instability may particularly be seen in Table 2 in the PR-AUC value of the Chain-of-Thought prompting method for DSC 33B.

6.2 Transformer Estimator Hyperparameters

Figure 6 and Figure 7 of Appendix G show the bar plots of Transformer estimator hyperparameter importance and correlation with PR-AUC, respectively. The importance plot highlights head 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 a 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 the 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 Effect of Various Factors on Hallucination Detection Quality

Table 2 shows that PR-AUC decreases as *model size* increases, which suggests that larger models produce more subtle hallucinations. The difficulty of hallucination detection may differ for LLMs from 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). According to Table 9 in Appendix F, the Transformer estimator does not significantly differ from the supervised methods for *general-purpose* LLMs like Llama 3 8B (0.639 PR-AUC for the Transformer detector vs. 0.631 PR-AUC for the MLP estimator). The results provided in Table 7 of Appendix D show that the Transformer estimator trained on both *domains* (code synthesis and code repair) generalizes well, while domain-specific training hurts cross-domain performance.

6.5 Efficiency of Transformer Estimator

The Transformer-based hallucination detector is *efficient* in terms of computational resources. Specifically, the estimator adds between 100M and 300M parameters (depending on architectural hyperparameters) to the LLM and takes less than an hour to train while showing the best quality of hallucination detection among the baselines. The overhead introduced by the estimator increases the end-to-end inference time by 10-30%. Table 3 shows the values of computational resources consumption estimated on testing subsets using a single A100 80GB GPU.

Resource	DSC 1.3B	DSC 6.7B	DSC 33B
GPU RAM (GB)	10	20	58
Training Time (min)	20 ± 8	18 ± 7	43 ± 29

Table 3: Computational resources consumption of the Transformer estimator during training. The values of GPU RAM consumption are the upper bounds. The training time is averaged.

6.6 Relation of Estimated Uncertainty to Functional Correctness

To evaluate how the estimated uncertainties relate to the functional correctness of generated

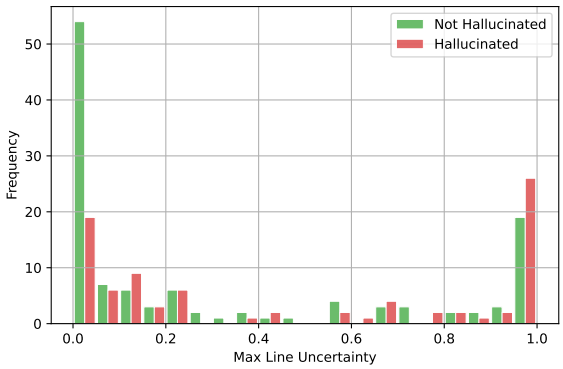

Figure 4: Histogram of maximum line uncertainties for hallucinated and correct functions.

code, we computed line-level uncertainty scores for code samples from the mixed dataset produced by DeepSeek-Coder-6.7B-Instruct. The uncertainty estimation was performed by the Transformer-based detector. For each function from the testing dataset, the estimated uncertainties were aggregated into a single value. We analyzed two types of aggregation: by taking the *mean* or the *maximum*. According to our experiments, both aggregations make sense, but the maximum of uncertainties relates more strongly to the functional incorrectness of the generated code. Figure 4 demonstrates the histogram of correct and hallucinated functions over maximum line uncertainties. In addition, Appendix H provides Figure 9 with a histogram of 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 for the presence of mistakes in the generated source code*.

6.7 Error Analysis of the Transformer-based Detector

To analyze the hallucination detection cases that are problematic for the Transformer-based detector, we used the estimated uncertainties of the lines of code generated by DeepSeek-Code 6.7B Instruct (see Figure 8 of Appendix H) to predict whether the entire code snippet is hallucinated. This is implemented by finding the *mean* or *maximum* of the uncertainties of the lines of code and thresholding it to obtain a binary prediction. We have chosen the standard threshold of 0.5 for binary classification; however, it may not be optimal. For *mean* aggregation, false negatives prevail with very few predictions of hallucinations (see Figure 11 in Appendix H), which indicates weak effectiveness of the mean aggregation with a 0.5 decision boundary. The histogram of averaged uncertainties for the lines of code is demonstrated in Figure 9 of Appendix H. For *maximum* aggregation, one can observe a balance between 36 false negative (FN) cases and 46 false positive (FP) cases in Figure 10 of Appendix H, which is more effective compared to the mean aggregation. Therefore, we focused on the manual error analysis of binary classification using the *maximum* aggregation of predicted uncertainties. Table 10 of Appendix H demonstrates a detailed analysis of problematic cases and their possible explanations for various error types and dataset difficulties. The results of the manual analysis demonstrate that the confusing or insufficiently detailed input prompts may lead to false negatives (cases of epistemic uncertainty), while false positives in the training dataset and multi-line statements in the generated code may lead to false positives (cases of aleatoric uncertainty). An additional interesting finding is that the detector may produce false positives if the generated code is not optimal and may need to be refactored. Such cases may demonstrate *potential applicability in code refactoring*.

7 Conclusion

We develop a supervised Transformer-based code 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. The experiments demonstrate that the estimated uncertainty provides a strong signal of functional incorrectness. The experimental results also highlight the increasing difficulty of detecting hallucinations in source code generated by larger LLMs. The error analysis also showed that hallucination detectors may potentially be useful for code refactoring. 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. Building upon the demonstrated ability to identify line-level functional incorrectness, a possible future work may be the integration of this hallucination detector into LLM-based coding agents. This would enable more autonomous and reliable agents capable of self-correction to move towards

more robust and trustworthy code.

Limitations

Due to the spelling sensitivity of the diff utility, the proposed diff-based annotation algorithm tends to mark the lines of generated code as hallucinated, even though the lines do not actually contain hallucinations. As discussed in Appendix C, the sensitivity introduces a labelling noise, but the conclusions of the paper are not affected by the noise.

Our detector primarily aims to support LLMs for in-domain tasks. Due to this, we do not claim full generalization to all existing languages and datasets. We consider developing a generalizable solution as future work.

The detector operates on internal representations of the LLM, such as hidden states and attention weights. Due to this, the detector cannot be directly applied to closed-source LLMs without access to the internal representations. Nevertheless, the detector allows to analyze hallucinations of black-box surrogates obtained via distillation.

The detector is LLM-specific, which means the detector trained for one specific LLM does not guarantee correct detection for another LLM. This limitation comes from the fact that various LLMs have internal representations distributed differently. Nevertheless, the training of detectors is very cheap and feasible for the majority of practitioners.

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.

Acknowledgments

The work of Georgii Andriushchenko, Roman Garaev, and Vladimir Ivanov was supported by the Ministry of Economic Development of the Russian Federation (agreement No. 139-10-2025-034 dd. 19.06.2025, IGK 000000C313925P4D0002).

References

Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2025. Codemirage: Hallucinations in code generated by large language models. *Preprint*, arXiv:2408.08333.

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

Amos Azaria and Tom Mitchell. 2023. The internal state of an LLM knows when it's lying. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 967–976, Singapore. Association for Computational Linguistics.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Yung-Sung Chuang, Linlu Qiu, Cheng-Yu Hsieh, Ranjay Krishna, Yoon Kim, and James R. Glass. 2024. Lookback lens: Detecting and mitigating contextual hallucinations in large language models using only attention maps. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1419–1436, Miami, Florida, USA. Association for Computational Linguistics.

Ekaterina Fadeeva, Roman Vashurin, Akim Tsvigun, Artem Vazhentsev, Sergey Petrakov, Kirill Fedyanin, Daniil Vasilev, Elizaveta Goncharova, Alexander Panchenko, Maxim Panov, Timothy Baldwin, and Artem Shelmanov. 2023. LM-polygraph: Uncertainty estimation for language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 446–461, Singapore. Association for Computational Linguistics.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Yarin Gal. 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, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

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. *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, Bangkok, Thailand. Association for Computational Linguistics.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *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*, ISSTA 2014, page 437–440, New York, NY, USA. Association for Computing Machinery.

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

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *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. *Preprint*, arXiv:2506.11021.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. Code llama: Open foundation models for code. *Preprint*, arXiv:2308.12950.

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

Artem Shelmanov, Ekaterina Fadeeva, Akim Tsvigun, Ivan Tsvigun, Zhuohan Xie, Igor Kiselev, Nico Daheim, Caiqi Zhang, Artem Vazhentsev, Mrinmaya Sachan, Preslav Nakov, and Timothy Baldwin. 2025. A head to predict and a head to question: Pre-trained uncertainty quantification heads for hallucination detection in LLM outputs. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 35712–35731, Suzhou, China. Association for Computational Linguistics.

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

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

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. In *Advances in Neural Information Processing Systems*, volume 37, pages 34188–34216. Curran Associates, Inc.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

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. Incoherence as oracle-less measure of error in llm-based code generation. *Preprint*, arXiv:2507.00057.

Roman Vashurin, Ekaterina Fadeeva, Artem Vazhentsev, Lyudmila Rvanova, Daniil Vasilev, Akim Tsvigun, Sergey Petrakov, Rui Xing, Abdelrahman Sadallah, Kirill Grishchenkov, 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*, 13:220–248.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Artem Vazhentsev, Ekaterina Fadeeva, Rui Xing, Gleb Kuzmin, Ivan Lazichny, Alexander Panchenko, Preslav Nakov, Timothy Baldwin, Maxim Panov, and Artem Shelmanov. 2025a. Unconditional truthfulness: Learning unconditional uncertainty of large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 35673–35694, Suzhou, China. Association for Computational Linguistics.

Artem Vazhentsev, Lyudmila Rvanova, Ivan Lazichny, Alexander Panchenko, Maxim Panov, Timothy Baldwin, and Artem Shelmanov. 2025b. Token-level density-based uncertainty quantification methods for eliciting truthfulness of large language models. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2246–2262, Albuquerque, New Mexico. Association for Computational Linguistics.

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).

A Unsupervised Uncertainty Quantification 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 N be 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:

$$U_{\text{msp}}(S) = -\sum_{i=1}^{N} \log P(t_i|t_1, t_2, ..., t_{i-1}) \quad (5)$$

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:

$$U_{\text{ppl}}(S) = -\frac{1}{N}\sum_{i=1}^{N} \log P(t_i|t_1, ..., t_{i-1}) \quad (6)$$

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

$$U_{\text{as}}(S) = -\frac{1}{H}\sum_{h=1}^{H} s_h \quad (7)$$

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

$$s_h = \sum_{i=1}^{T} \log\left(\left[A_h^l\right]_{ii}\right) \quad (8)$$

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:

$$U_{\text{mte}}(S) = \max_{t \in S} H_t \quad (9)$$

where H_t is a token entropy calculated as

$$H_t = -\sum_{v \in V} p_{t,v} \cdot \log p_{t,v} \quad (10)$$

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 Details of Line-level Code Hallucination Detection

B.1 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:

$$F_{\text{hs}}(t) = h_l(t) \quad (11)$$

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:

$$F_{\text{aw}}(t_i) = \left\{\alpha_{i,i-j}^{q,l}\right\}_{j=1}^{k} \quad (12)$$

for all $q \in \{1, \ldots, Q\}$, $l \in \{1, \ldots, L\}$, where k is the 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:

$$F_{\text{tp}}(t_i) = \left\{\log P(t \mid t_{<i}, \mathbf{x})\right\} \quad (13)$$

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

$$F_{\text{ts}}(t_i) = \{S^l(t_i)\}_{l=1}^{L-1} \quad (14)$$

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 $l+1$

$$S^l(t_i) = \{\cos(E_{w_1}, E_{w_2})\} \quad (15)$$

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 $l = 1, 2, \ldots, L$:

$$F_{\text{or}}(t_i) = \left\{ \left(R^l(t_i) \right)^{-1} \right\}_{l=1}^{L} \tag{16}$$

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.

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

$$F_{\text{lbl}}(t_i) = \left\{ LR^{q,l}(t_i) \right\} \tag{17}$$

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

$$LR^{q,l}(t_i) = \frac{A^{q,l}_{\text{context}}(t_i)}{A^{q,l}_{\text{context}}(t_i) + A^{q,l}_{\text{gen}}(t_i)} \tag{18}$$

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.

B.2 Transformer Estimator

The detection using the Transformer estimator is organized in the following manner. First, a feature vector $F(t)$ is produced for each token t in the generated source code $\mathbf{y}$. The feature vector $F(t)$ is then projected to vector $\tilde{f}_t$ using a fully-connected layer FC_{proj} to be able to pass it to the Transformer encoder:

$$\tilde{f}_t = FC_{\text{proj}}(F(t)) \tag{19}$$

Then, each projected feature vector $\tilde{f}_t$ for all $t \in s$ is added to the trainable line position embeddings E_s specific for each line to obtain a contextualized feature vector $\tilde{f}_{t,s}$:

$$\tilde{f}_{t,s} = \tilde{f}_t + E_s \tag{20}$$

Next, the feature vector $\tilde{f}_{t,s}$ is passed to Transformer encoder T and averaged to obtain contextualized vector representation h_s of token $t \in s$:

$$h_s = \frac{1}{|s|} \sum_{t \in s} T(\{\tilde{f}_{t,s}\}_{t \in \mathbf{x} \circ \mathbf{y}}) \tag{21}$$

The uncertainty for a line of code s is estimated using a multi-layer perceptron MLP_{clf} and sigmoid activation function σ:

$$U(s) = \sigma\left(\text{MLP}_{\text{clf}}(h_s)\right) \tag{22}$$

To prevent overfitting, the dropout regularization for MLP_{clf} is provided.

B.3 Supervised Baseline Estimators

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:

$$U_{\text{lr}}(S) = \text{f}_{\text{lr}}\left(\frac{1}{N} \sum_{i=1}^{N} F(t_i) \right) \tag{23}$$

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

$$U_{\text{mlp}}(S) = \frac{1}{N} \sum_{i=1}^{N} f_{\text{mlp}}\left(h(t_j)\right) \tag{24}$$

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.

C Quality of Diff-based Dataset Annotation

We evaluated the quality of the diff-based automatic annotation with respect to the manually annotated dataset. Specifically, we calculated balanced accuracy, precision, recall, F1, and Matthews correlation coefficient (MCC) between diff-based and manually labeled testing subsets of code generated by DeepSeek-Coder 6.7B Instruct (see Table 1 (Mixed dataset) and Table 5 for statistics). Table 4 presents the annotation quality metrics for diff-based and GPT-based labelling. The metrics show that the diff-based annotation provides non-random labels. However, the metrics also highlight a substantial error between the manual labels and the labels obtained with diff. We claim that the substantial error between the labels obtained with diff and the manual labels is introduced by the frequent *differences in syntactic constructs* (e.g., variable naming, formatting) between the generated and the ground-truth lines of code that have the same semantics (see Figure 5 for an example).

To further assess whether this level of error affects our conclusions, we conducted an additional

Generated Code

```
1 def cube_Sum(n):
2     sum = 0
3     for i in range(1, n+1):
4         sum += (2 * i) ** 3
5     return sum
```

Ground-truth Code

```
1 def cube_Sum(n):
2     num = 0
3     for i in range(1, n + 1):
4         num += (2 * i) ** 3
5     return num
```

Figure 5: Example of diff-based annotation errors introduced by differences in syntactic constructs of the lines of code (highlighted in red). Differences in *variable naming* make the annotation falsely label Lines 2 and 5 as hallucinating. Differences in *formatting* make the annotation falsely label Line 3 as hallucinating.

robustness experiment using an alternative annotation strategy (see Table 4). Specifically, we prompted GPT-5.2 to annotate the dataset of code generated by DeepSeek-Coder 6.7B Instruct. A quantitative validation against manual labels shows substantially higher agreement. This indicates that GPT-assisted annotation still contains small but non-zero noise, while being considerably closer to manual labeling. Due to the capabilities of LLMs to analyze the semantic similarity of the lines of code, the GPT-based annotation substantially reduces the noise introduced by the diff-based annotation.

Metric	Random	Diff-based	GPT-based
MCC	.392	.664	.940
Accuracy	.389 $\pm$.032	.674 $\pm$.028	.922 $\pm$.016
Precision	.510 $\pm$.031	.715 $\pm$.027	.976 $\pm$.010
Recall	.510 $\pm$.031	.749 $\pm$.026	.928 $\pm$.016
F1	.510 $\pm$.027	.730 $\pm$.021	.951 $\pm$.010

Table 4: Annotation quality metrics calculated with respect to the manually labeled testing subset of code generated by DeepSeek-Coder 6.7B Instruct. Precision, recall, and F1 metrics are calculated with respect to the positive class. The balanced accuracy is measured.

We used the training subset of the GPT-annotated dataset to train with hyperparameter optimization the notable detectors from Table 2 and tested the detectors on the testing subset. The resulting PR-AUC values shown in Table 6 are consistent with those reported in Table 2 in terms of relative ranking and performance trends across methods. Although absolute values vary slightly, the ordering of detectors and the overall conclusions remain unchanged. These results suggest that our findings are stable under substantially cleaner labeling. While the diff-based annotation introduces non-negligible noise, the additional experiment indicates that this noise does not materially alter the conclusions of the study.

Statistic Name	Value
Number of examples	205
Number of lines	1359
Number of hallucinated lines	262 (19%)
Number of correct lines	1097 (81%)

Table 5: Statistics of the manually annotated dataset (testing subset) of code generated by DeepSeek-Coder 6.7B Instruct on which we calculated the annotation performance metrics.

Estimator	Features	DSC 6.7B
Random	N/A	.174 $\pm$.025
Prompting	Zero-shot prompt	.273 $\pm$.028
Prompting	One-shot prompt	.263 $\pm$.029
Prompting	CoT prompt	.260 $\pm$.051
CodeBERT	Tokens	.340 $\pm$.040
Attention Score	Attention weights	.336 $\pm$.045
MLP	Hidden States	.477 $\pm$.063
Transformer	TP, AW, HS	**.499** $\pm$.061

Table 6: PR-AUC values computed on code generated by DeepSeek-Coder 6.7B Instruct and annotated by GPT-5.2. The abbreviations “TP", “AW", and “HS" stand for token probabilities, attention weights, and hidden states respectively.

D Transformer Estimator Performance for Various Task Domains

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

Table 7: Transformer-based hallucination detector performance for various task domains. The highest values are highlighted in bold, the second-highest values are underlined. The standard deviation is obtained through a statistical bootstrap.

E 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 8: 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.

F UQ Metrics for a General-purpose LLM

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

Table 9: 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.

G Effect of Transformer Estimator Hyperparameters on PR-AUC metric

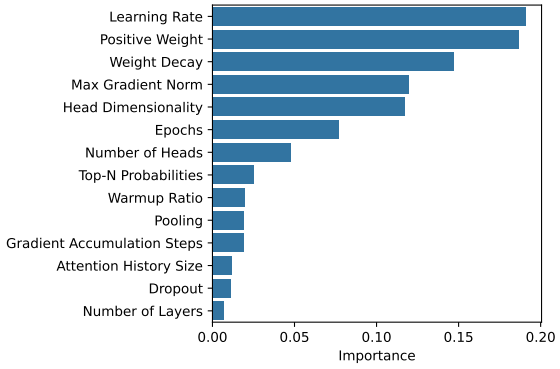

Figure 6: Transformer estimator hyperparameters importance relative to PR-AUC metric

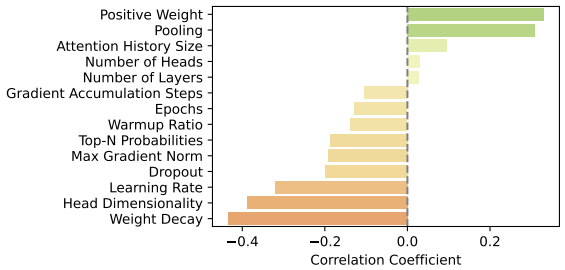

Figure 7: Transformer estimator hyperparameters correlation with PR-AUC metric

H Error Analysis of Transformer Estimator

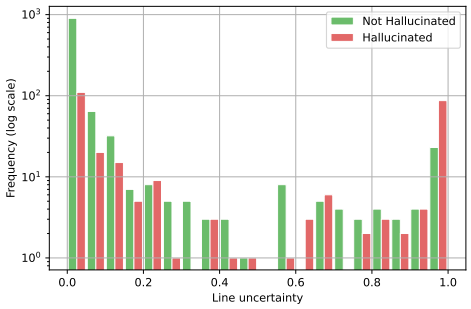

Figure 8: Histogram of line uncertainties for hallucinated and correct lines of code.

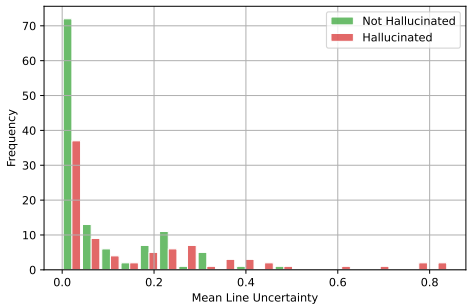

Figure 9: Histogram of mean line uncertainties for hallucinated and correct functions.

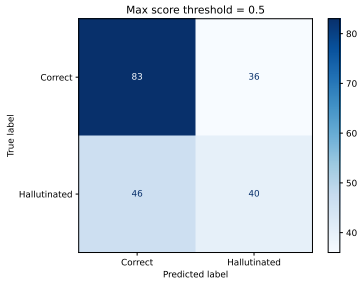

Figure 10: Confusion matrix of hallucinated function prediction via maximum uncertainty of the code lines.

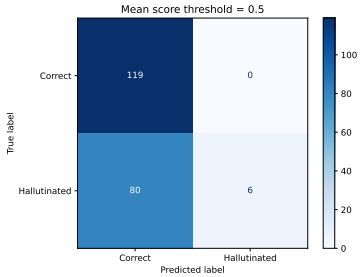

Figure 11: Confusion matrix of hallucinated function prediction using mean uncertainty of the lines of code.

Dataset (Difficulty)	Error Types	Frequency	Problematic Cases	Error Reason Explanation	Uncertainty Type
MBPP (Easy)	FN	27%	Code related to input cases and output formats.	The detector misses hallucinations due to the lack of necessary information in the prompt.	Epistemic
	FP	21%	The generated code is imperfect or contains excessive parts.	The detector detects false hallucinations if the generated code is imperfect and may be improved.	Aleatoric
HumanEval (Medium)	FN	30%	Incorrectly generated code due to misunderstanding of the task.	The detector misses hallucinations due to the confusing or not sufficiently detailed prompt.	Epistemic
	FP	10%	"If" clauses and single long statements distributed over multiple lines.	Imperfect training data ("if" clauses are often incorrectly labelled as hallucinating) and failing to handle long statements distributed over multiple lines.	Aleatoric
HumanEval Java (Medium)	FN	11%	Complex bugs related to data structures and logic.	The LLM weakly understands the essence of the bug.	Epistemic
	FP	32%	Multi-line statements, "if" clauses, and other lines.	Failing to handle multi-line statements and imperfect training data.	Aleatoric
Defects4J (Hard)	FN	17%	Complex bugs related to data structures, program logic, insufficiently detailed prompt.	The LLM does not fully understand the bug, or the prompt lacks important details.	Epistemic
	FP	12%	Complex bugs related to data structures, program logic, "if" clauses	The LLM does not fully understand the bug, imperfect training data	Epistemic, Aleatoric

Table 10: Error analysis of Transformer estimator on code generated by DeepSeek-Coder 6.7B Instruct. The error types column includes false negative (FN) and false positive (FP).

I 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 11: 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.

J 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 12: 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 provide percentage information relative to the *Total Lines* column in the parentheses.

K 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 12: 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 13: Example of the code repair prompt for DeepSeek-Coder. The prompt is inspired by the prompts proposed in Collu-Bench (Jiang et al., 2024).
```

L 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 (||).

M Prompting Baselines Prompts

```
TASK
{task formulation}

SOLUTION:
{generated code}

For each line of SOLUTION, check whether the line correct or buggy and may lead to failing tests.
Label each line with 1 if the line is buggy or with 0 if the line is correct
In your response, provide ONLY binary labels (0 or 1) for each line of SOLUTION in the following
format:
Line 1: <Label>
Line 2: <Label>
...
```

Figure 15: Zero-shot prompting template for code synthesis tasks.

```
PROBLEM DESCRIPTION
[task description]

FUNCTION
{function with buggy lines of code}

BUGGY CODE
{buggy lines of code}

FIXED BUGGY CODE
[generated lines of code]

For each line of FIXED BUGGY CODE, check whether the line is fixed correctly or incorrectly and may
fail tests.
Label each fixed line with 1 if the fixed line is incorrect or with 0 if the fixed line is correct.
In your response, provide ONLY binary labels (0 or 1) for each line of FIXED BUGGY CODE in the
following format:
Line 1: <Label>
Line 2: <Label>
...
```

Figure 16: Zero-shot prompting template for code repair tasks.

````
TASK
Write a python function to check whether the count of divisors is even or odd.

SOLUTION:
Line 1: ```python
Line 2: import math
Line 3: def count_divisors(n):
Line 4:     result = 0
Line 5:     for i in range(1, int(math.sqrt(n)) + 1):
Line 6:         if n % i == 0:
Line 7:             if n / i == i:
Line 8:                 result += 1
Line 9:             else:
Line 10:                 result += 2
Line 11:     return (result % 2) == 0
Line 12: ```


For each line of SOLUTION, check whether the line correct or buggy and may lead to failing tests.
Label each line with 1 if the line is buggy or with 0 if the line is correct
In your response, provide ONLY binary labels (0 or 1) for each line of SOLUTION in the following
format:
Line 1: <Label>
Line 2: <Label>
...

LABELS:
Line 1: 0
Line 2: 0
Line 3: 0
Line 4: 0
Line 5: 0
Line 6: 0
Line 7: 1
Line 8: 0
Line 9: 0
Line 10: 0
Line 11: 0
Line 12: 0

TASK
{task formulation}

SOLUTION:
{generated code}

For each line of SOLUTION, check whether the line correct or buggy and may lead to failing tests.
Label each line with 1 if the line is buggy or with 0 if the line is correct
In your response, provide ONLY binary labels (0 or 1) for each line of SOLUTION in the following
format:
Line 1: <Label>
Line 2: <Label>
...
````

Figure 17: One-shot prompting template for code synthesis tasks.

````
PROBLEM DESCRIPTION
None

FUNCTION
```java
public Map<String, Integer> getHeaderMap() {

<bug>
    return new LinkedHashMap<String, Integer>(this.headerMap);
</bug>
}


BUGGY CODE
```java
    return new LinkedHashMap<String, Integer>(this.headerMap);
```

FIXED BUGGY CODE
Line 1: ```java
Line 2:     return new HashMap<String, Integer>(this.headerMap);
Line 3: ```


For each line of FIXED BUGGY CODE, check whether the line is fixed correctly or incorrectly and may
fail tests.
Label each fixed line with 1 if the fixed line is incorrect or with 0 if the fixed line is correct.
In your response, provide ONLY binary labels (0 or 1) for each line of FIXED BUGGY CODE in the
following format:
Line 1: <Label>
Line 2: <Label>
...

LABELS:
Line 1: 0
Line 2: 1
Line 3: 0

PROBLEM DESCRIPTION
[task description]

FUNCTION
{function with buggy lines of code}

BUGGY CODE
{buggy lines of code}

FIXED BUGGY CODE
[generated lines of code]

For each line of FIXED BUGGY CODE, check whether the line is fixed correctly or incorrectly and may
fail tests.
Label each fixed line with 1 if the fixed line is incorrect or with 0 if the fixed line is correct.
In your response, provide ONLY binary labels (0 or 1) for each line of FIXED BUGGY CODE in the
following format:
Line 1: <Label>
Line 2: <Label>
...
````

Figure 18: One-shot prompting template for code repair tasks.
````

```
TASK
{task formulation}

SOLUTION:
{generated code}

For each line of SOLUTION, check whether the line correct or buggy and may lead to failing tests.
Label each line with 1 if the line is buggy or with 0 if the line is correct.
In your response, provide a CHAIN-OF-THOUGHT first containing the label and explanation for each line
of SOLUTION.
Then, provide BINARY LABELS (0 or 1) for each line of SOLUTION in the following format:
Line 1: <Label>
Line 2: <Label>
...

CHAIN-OF-THOUGHT:
```

Figure 19: Chain-of-Thought prompting template for code synthesis tasks.

```
PROBLEM DESCRIPTION
[task description]

FUNCTION
{function with buggy lines of code}

BUGGY CODE
{buggy lines of code}

FIXED BUGGY CODE
[generated lines of code]

For each line of FIXED BUGGY CODE, check whether the line is fixed correctly or incorrectly and may
fail tests.
Label each fixed line with 1 if the fixed line is incorrect or with 0 if the fixed line is correct.
In your response, provide a CHAIN-OF-THOUGHT first containing the label and explanation for each line
of SOLUTION.
Then, provide BINARY LABELS (0 or 1) for each line of SOLUTION in the following format:
Line 1: <Label>
Line 2: <Label>
...

CHAIN-OF-THOUGHT:
```

Figure 20: Chain-of-Thought prompting template for code repair tasks.