Hallucinations in Code Change to Natural Language Generation: Prevalence and Evaluation of Detection Metrics

Anonymous ACL submission

Abstract

Language models have shown strong capabilities across a wide range of tasks in software engineering, such as code generation, yet they suffer from hallucinations. While hallucinations have been studied independently in natural language and code generation, their occurrence in tasks involving code changes which have a structurally complex and context-dependent format of code remains largely unexplored. This paper presents the first comprehensive analysis of hallucinations in two critical tasks involving code change to natural language generation: commit message generation and code review comment generation. We quantify the prevalence of hallucinations in recent language models and explore a range of metric-based approaches to automatically detect them. Our findings reveal that approximately 50% of generated code reviews and 20% of generated commit messages contain hallucinations. Whilst commonly used metrics are weak detectors on their own, combining multiple metrics substantially improves performance. Notably, model confidence and feature attribution metrics effectively contribute to hallucination detection, showing promise for inference-time detection.[1]

1 Introduction

AI-based software engineering tools are becoming increasingly ubiquitous due to their potential to improve developer productivity (Jain et al., 2022; Fan et al., 2023; Hou et al., 2024). While such tools can accelerate software development, their reliance on underlying language models exposes the risk of hallucination—the phenomenon where models generate outputs that are inconsistent with their inputs or fabricate non-existent information (Ji et al., 2023; Huang et al., 2025). Such behavior may decrease developer productivity or even mislead junior developers (Ferino et al., 2025), allowing errors to propagate through to the software. Although prior research has focused on the effects of hallucination during code generation (Liu et al., 2024; Tian et al., 2024; Agarwal et al., 2024), these effects remain largely unexplored in generation tasks involving code changes. Unlike complete code files, code changes present snippets of both the old and new versions simultaneously, which could potentially amplify hallucinations due to the model's need to process and reason about multiple code states with partial context.

Indeed, code changes commonly used in the software engineering workflows (Tao et al., 2012; Grazia et al., 2023). Recent work also leveraged code changes as primary inputs of language models for automated software engineering tasks such as code reviews (Li et al., 2022; Lin et al., 2023). Given the increasing use of code changes in generation tasks, there is a need to understand the prevalence and effectiveness of the current detection metrics. The fragmented and context-dependent nature of code changes may increase hallucination risk and hinder detection.

In this paper, we present a comprehensive study of hallucinations in code change to natural language (`CodeChange2NL`) generation tasks. We focus on two key tasks: (1) automated commit message generation, which aids developers in documenting what and why code was changed, and (2) automated code review generation, which assists reviewers in identifying potential issues in code changes and suggesting improvements. To systematically analyze hallucination in `CodeChange2NL`, we first develop a hallucination annotation workflow specific to the `CodeChange2NL` context based on the outputs from task-specific models. We then empirically evaluate the effectiveness of various metric-based approaches for automatically detecting these hallucinations. In particular, we examine both reference-based metrics (which compare against human-written references) and reference-free metrics (without the references).

[1] All code and data will be released upon acceptance.

Our findings reveal the severity of the hallucination problem in CodeChange2NL tasks. We found that nearly 50% of model-generated code reviews and 20% of generated commit messages contain hallucinations. The three predominant categories of hallucinations are input inconsistency (where the generated NL is inconsistent with the code change), logic inconsistency (where the NL contains internally contradictory reasoning), and intention violation (where the generation fails for the specific task, e.g., it is not a review comment for code review but just a summary of the code change). Furthermore, we demonstrate that individual metrics for hallucination detection perform only marginally better than random chance (56.6% ROC-AUC for code review and 61.7% for commit messages). However, combining multiple metrics yields substantial improvements (69.1% and 75.3% respectively). Notably, reference-free metrics show promising results comparable to using all available metrics, suggesting the feasibility of detecting hallucinations without ground truth references.

This work makes three primary contributions: (1) the first systematic characterization of hallucinations in code change to natural language tasks, revealing the severity and patterns of the problem; (2) a comprehensive evaluation of automatic hallucination detection methods, demonstrating that combining multiple metrics significantly improves detection capability; and (3) identification of key reference-free metrics (model confidence and attribution scores) that effectively predict hallucinations, facilitating real-time detection in production environments without requiring reference text.

2 Related Work

Hallucination in Natural Language Generation Initially, Maynez et al. (2020) categorized hallucinations in summarization into two types: **intrinsic** hallucinations (where models misinterpret information present in the input, generating content that contradicts the source document) and **extrinsic** hallucinations (where models forge information absent from the input that cannot be verified using available information). Recently, Huang et al. (2025) identified three subcategories of intrinsic hallucinations in LLMs: instruction-inconsistent (outputs are not consistent with the instruction), logic inconsistency (output itself exhibits internal logical contradictions), and context inconsistency (outputs are not consistent with the provided input context). Huang et al. (2025) further refined these factual hallucinations by distinguishing between factual contradiction (outputs that can be grounded but contradict real-world knowledge) and factual fabrication (outputs that are completely made up with no basis in reality or verifiable facts). Research on hallucination in code generation tasks also grounds hallucination types based on these categories (Liu et al., 2024). This taxonomy aligns closely with our CodeChange2NL tasks and serves as a foundation to determine the hallucination types in Section 3.2.

Hallucination in Code to Natural Language Generation Different from hallucination research in natural language to code generation, which primarily focuses on incorrect code generations e.g., dead-/unreachable code, syntactic incorrectness (Liu et al., 2024; Agarwal et al., 2024), hallucination in code to natural language generation focuses on natural language utterances that are incorrect with respect to the code/task at hand. Whilst many hallucinations in code generation can be verified by static analysis and execution (Tian et al., 2024), these solutions are not applicable for natural language outputs. Recent work examined hallucination in code-to-natural language tasks (Zhang, 2024; Maharaj et al., 2024; Kang et al., 2024). However, they primarily focus on compilable code implementations (e.g., the full body of a method). For example, Maharaj et al. (2024) studied entity-level hallucination in code summarization, where the input consists of a method-level function containing adequate contextual information. Yet, other code-to-natural language tasks involving snippets of code changes remain largely overlooked, despite their common use in real-world scenarios like commit message generation and code review (Lin et al., 2023). Moreover, due to the technical constraints of long-context modeling, snippets of code changes are often used as inputs for generation tasks instead of the complete code context (Lu et al., 2025; Berabi et al., 2024). The fragmented, context-dependent nature of code changes may increase hallucination risk and hinder detection, motivating our investigation into their prevalence and the effectiveness of existing metrics.

Automatic Hallucination Detection Automatic hallucination detection methods fall into two broad categories: reference-based and reference-free. *Reference-based* metrics use ground truth to gauge the quality of the generated outputs, using this quality as an estimation of hallucination. This in-

cludes lexical overlap such as BLEU (Papineni et al., 2002), which evaluates n-gram similarity between generated and reference texts. This is widely used in both Code2NL and NL2NL tasks (Liu et al., 2018a; Tufano et al., 2021; Li et al., 2022; Liu et al., 2025). More advanced metrics use Natural Language Inference (NLI): the model output is treated as a "hypothesis" to be validated against the reference. An entailment classifier labels output as entailment or contradiction, which maps to faithful or hallucinated content (Manakul et al., 2023; Elaraby et al., 2023; Hu et al., 2024; Valentin et al., 2024). *Reference-free* methods operate in many open-ended generation settings, where a reference is unavailable, by analyzing internal model behaviors and input-output relationships. One family of approaches estimates uncertainty inside models during generation (Guerreiro et al., 2023; Huang et al., 2024), with hallucinations typically exhibiting lower confidence in probability distributions and higher entropy. Another promising line is feature attribution techniques (Tang et al., 2022; Chen et al., 2025), which examine how inputs influence outputs, e.g., when a model hallucinates, its attention patterns or hidden states behave anomalously. While these metrics have been used to detect hallucinations in various NL2NL tasks, such as machine translation and question answering (Guerreiro et al., 2023; Dale et al., 2023), their capabilities in `CodeChange2NL` tasks remain unknown.

3 Study Design

3.1 Research Questions

RQ1: To what extent do task-specific language models hallucinate in code change to natural language tasks? Prior work on hallucination in software engineering has focused on code generation, which can be verified deterministically. However, little attention has been paid to hallucinations in `CodeChange2NL` generation tasks, such as code review comment generation and commit message generation.

RQ2: How effectively can existing hallucination detection methods perform on code change to natural language tasks? While prior work in NLP have developed various methods (Dale et al., 2023; Huang et al., 2025; Ji et al., 2023) to detect hallucinations in natural language generation, their applicability to the bi-modal scenario of `CodeChange2NL` remains unknown. Effective detection in such contexts requires an understanding of the semantics behind both code, natural language, and their interaction.

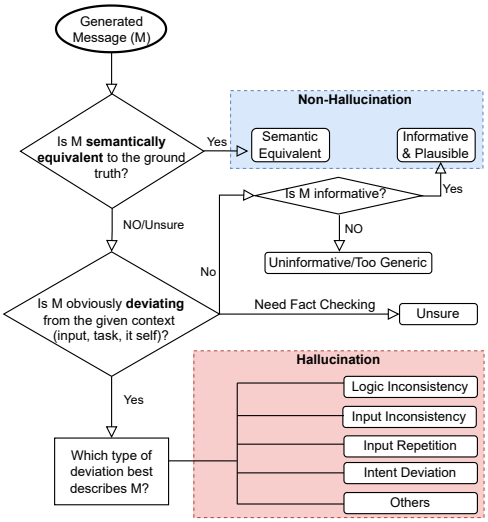


Figure 1: Hallucination Annotation Flowchart

3.2 Hallucination Annotation Workflow

Since no existing work addresses hallucinations in the `CodeChange2NL` context, we developed a decision-tree-based hallucination detection workflow by adapting taxonomies from both code generation (Liu et al., 2024) and natural language hallucination (Huang et al., 2025). Our workflow[2] (see Figure 1) evaluates a generated NL as follows:

Semantic Equivalence. We first determine whether the generated NL is semantically equivalent to the ground truth (i.e., conveying the same intent with similar framing and emphasis). If equivalent, the output is classified as non-hallucination.

Contextual Faithfulness. For semantically non-equivalent outputs, we assess whether the NL deviates from the context (source code, task specification, and generated text itself). Non-deviating outputs are classified as either *Informative & Plausible* (valid alternatives) or *Uninformative* (truisms).

Hallucination Type Classification. When context deviation exists, we categorize the hallucination into five types:[3] 1) *Input Inconsistency*, where the generation conflicts with the source code, e.g., pointing out a non-existent issue in code review or speculating intent that contradicts the code change in commit messages; 2) *Logic Inconsistency*, where the generation is internally illogical, independent of the input; 3) *Input Repetition*, where the generation directly copies from the input; 4) *Intent Deviation*,

[2]See Appendix A for definition and annotation guidelines.
[3]Examples are provided in Appendix A.3.

where the generation deviates from the task's goal, e.g., not identifying issues in a code review or not explaining the code change in a commit message; and 5) *Others* for cases that are not covered by the above types. Cases requiring additional project specific fact-checking are labeled as *Unsure*.

3.3 Datasets and CodeChange2NL Generation

Datasets. We choose the widely used CodeReviewer (Li et al., 2022) dataset for code review comment generation and CommitBench (Schall et al., 2024) for commit message generation. The CodeReviewer corpus contains code diff and natural language review pairs, across 9 popular programming languages and over 1k GitHub projects. It includes 118k training, 10k validation, and 10K testing examples. CommitBench contains code diffs paired with natural language commit messages, spanning over 72k GitHub repositories and 6 programming languages. It includes 1.16 million training examples and 250k examples each for validation and testing. While related, the two tasks are different in nature—commit messages are primarily descriptive, whereas code reviews require deeper reasoning about functional correctness and potential impacts across the codebase.

Models. To analyze hallucination behaviors, we conduct experiments to select language models that are highly capable in both tasks. This is determined by BLEU-4 results, which is the most commonly used metric (Li et al., 2022; Schall et al., 2024). We choose two recent LLM families (Qwen2.5 and Llama3.1)[4] with varied model sizes for both direct prompting (7-8B, 70-72B) and task-specific fine-tuning (7-8B). We also fine-tune CCT5 (Lin et al., 2023), which is a 220M T5-based model pre-trained on 1.5M code change to commit message pairs. We used the original training data in two datasets to fine-tune the models. We found that fine-tuned models performed the best for both tasks.[5] Table 1 (Overall columns) presents the experimental results. Thus, we select the three fine-tuned models to generate outputs for hallucination analysis in Sections 4 and 5.

3.4 Hallucination Detection Methodology

We use both reference-based and reference-free hallucination detection approaches: the former for model development where the ground truth is available, and the latter for real-world deployment where references are unavailable. Table 2 presents a summary of the metrics we used, including two types of reference-based (BLEU-4 and NLI), and three types of reference-free (similarity, uncertainty, and feature-attribution). Uncertainty and feature-attribution metrics are calculated with either LLaMA3.1-8B-Instruct (Grattafiori et al., 2024), Qwen2.5-7B-Instruct (Yang et al., 2025) or CCT5 (Lin et al., 2023). Due to space limitations, detailed descriptions and formulas are provided in Appendix D.1. In total, 26 unique methods were considered: 2 reference-based metrics + 3 similarity scores + 3 models × 7 feature attribution and uncertainty metrics.

Model	CodeReview		CommitBench	
	Overall	Sample	Overall	Sample
Llama3.1-8B	5.28	5.25	15.06	15.29
Qwen2.5-7B	5.43	5.73	15.37	15.57
CCT5	**5.58**	**6.53**	**17.45**	**17.46**

Table 1: Performance (BLEU-4 in %) of fine-tuned models on CodeReview and CommitBench benchmarks.

4 To what extent do task-specific language models hallucinate in CodeChange2NL tasks?

To address RQ1, we manually categorize the messages generated by the three fine-tuned models into our CodeChange2NL hallucination annotation workflow introduced in Section 3.2 to identify the presence and types of hallucinations. Using the annotated samples, we further analyze the overall prevalence of hallucinations and their distributional patterns across models and two datasets.

4.1 Manual Annotation

We selected the top 3 fine-tuned models (lama3.1-8B, Qwen2.5-7B, and CCT5) to generate messages in the test set. To address RQ1, we manually labeled a subset of samples that were randomly selected from the test set of each task, constituting a statistically significant sample size with a confidence level of 90% and a margin of error of ±5%. This results in 264 samples for CodeReviewer comments and 268 samples for CommitBench. In total, we annotated 1,596 samples, including 264 × 3 model outputs for CodeReviewer comments and 268 × 3 for CommitBench messages.

Two annotators (authors of the paper) with 5+ years of experience in computer science and software engineering annotated all samples. We con-

[4]These were the latest models at the time of experiment.
[5]See Appendix B for details on prompting and fine-tuning.

Metric	Type	Description
BLEU-4	Lexical-Overlap	The n-gram overlap between the generation y and reference $\hat{y}$.
Entailment	NLI	The probability that a NLI classifier predicts $\hat{y}$ entails y. We used nli-deberta-v3[6] as the classifier.
Similarity	Similarity	The embedding-based cosine similarity between the generation y and source code x. We used three embeding models: codebert-base[7], codet5p-220m-bimodal[8], and codet5p-770m[9].
SeqLogProb	Uncertainty	The average negative log-probability of the generated tokens in y as assigned by a language model M.
SeqLogit	Uncertainty	The average raw logit score (pre-Softmax) of the generated tokens in y from a model M.
SeqEntropy	Uncertainty	The average entropy of the generated tokens in y from a model M.
Source Attribution	Feature Attribution	The average of the maximum attribution scores from source tokens to each generated token in y (i.e., $\frac{1}{T}\sum_{t=1}^{T}\max_{i\in[1,N]} A_{i,t}$, where $A_{i,t} = x_i \times \frac{\partial y_t}{\partial x_i}$ is the importance of x_i to y_t from a model M). A higher score represents source contributes more strongly to y.
Target Attribution	Feature Attribution	The average of the maximum attribution scores from previously generated tokens $(y_1, \ldots, y_{t-1})$ to each current token y_t. A higher score represents the reliance on previously generated tokens.
Changed Attribution	Feature Attribution	The average of the maximum attribution scores from source tokens that are changed (in +, - lines) to each generated token in y. A high score represents changed tokens contributes strongly to y.
Unchanged Attribution	Feature Attribution	The average of the maximum attribution scores from source tokens that are unchanged to each generated token in y. A high score represents unchanged snippets in source contributes strongly to y.

Table 2: Descriptions of hallucination detection metrics, including into *reference-based* (BLEU-4 and Entailment) and *reference-free* (all others). For uncertainty and feature attribution, the model $M \in \{\text{LLaMA3.1-8B}, \text{Qwen2.5-7B}, \text{and CCT5}\}$. We apply both self-attribution (generator attributes its own output) and cross-attribution (external model attributes generator's output). See Appendix D.1 for a detailed description.

Category	Type	CodeReviewer			CommitBench		
		CCT5	Llama3.1	Qwen2.5	CCT5	Llama3.1	Qwen2.5
Non-Hallucination	Semantic_Equivalent	1.5	1.1	1.5	11.2	12.3	16.4
	Informative	9.5	9.8	8.7	48.1	42.5	44.4
Uninformative	Uninformative	20.1	1.5	3.8	15.7	7.1	9.7
Unsure	Unsure	22.0	41.3	43.2	5.6	16.4	15.3
Hallucination	Input_Inconsistency	26.5	23.9	24.6	17.2	19.8	13.1
	Input_Repetition	4.2	0.0	0.0	0.0	0.7	0.7
	Intent_Deviation	0.8	17.4	15.9	0.4	0.4	0.0
	Logic_Inconsistency	14.0	4.5	1.9	1.9	0.7	0.4
	Others	1.5	0.4	0.4	0.0	0.0	0.0
Total Hallucination		47.0	46.2	42.8	19.5	21.6	14.2

Table 3: The distribution (percentage) of hallucination categories and types for annotated samples. The Category column is the high-level category in Figure 1. The "Total Hallucination" is the sum of the four hallucination types.

ducted two pilot rounds (150 samples each) to refine the taxonomy and guidelines. Cohen's κ improved from 0.36/0.30 (CodeReviewer/CommitBench) in the first round to 0.56/0.38 in the second. Final disagreements were resolved through discussion, achieving near-perfect agreement ($\kappa = 0.98$ / 0.96). The annotators then divided the remaining samples (half-half), cross-examining each other's work to ensure consistent labeling.

4.2 Hallucination Prevalence and Patterns

Table 3 shows that hallucination rates vary significantly across tasks. For the code review task, all models exhibit high hallucination rates ranging from 42.8% to 47.0%. Surprisingly, although CCT5 achieves the highest BLEU score on the CodeReviewer dataset among the three models (Table 8), it also exhibits the highest hallucination rate at 47.0%. This highlights the risk of hallucinations even in models with strong BLEU performance. On the other hand, the commit message generation task has a lower hallucination rate than code review (14.2% to 21.6%), where Qwen2.5 has the lowest rate at 14.2%. This may be because code review is more challenging than commit message generation, as it requires identifying problems and providing specific feedback beyond what is directly observable in the code changes. Such added complexity might lead to increased hallucination behavior.

The overall distribution of hallucination types varies between tasks. Notably, the Input Inconsistency emerges as the dominant hallucination type for both tasks. This suggests that models frequently generate messages that contradict or misrepresent the actual code changes. One frequent issue in code review is that the generated messages tend to fabricate non-existent code tokens. For example, CCT5 suggests *"I think this should be* `orderPath` *instead*

of `orderPathKey`". However, `orderPathKey` does not appear in the code change:[10] `+~public static final String ORDER\_PATH = "orderPath";` This suggests that the model does not fully understand the meaning of newly introduced code. In the commit message task, models also often misunderstand the code changes. For example, the generated message *"nomad: fix peers.json recovery for protocol version 3"* misrepresents the change, which actually adds support for Nomad versions **below 3**, as indicated by the code line `+ if s.config.RaftConfig .ProtocolVersion < 3 {`.[11]

Intent deviation and logic inconsistency appear as another two pronounced hallucination types in the code review task, but they are rare in the commit message generation, suggesting that commit message generation models generate messages that better align with the task and suffer less logic inconsistency. Interestingly, we observe many cases where the generated review comment reads more like a commit message—for example, *"This is a temporary fix."*, which describes the code change rather than providing a review.

Different models exhibit different type of hallucinations. CCT5, which is the specialized fine-tuned model demonstrates higher logic inconsistencies (14.0% in CodeReviewer) but significantly lower intent deviation (0.8%) than general-purpose LLMs. On the other hand, larger models (Llama3.1, Qwen2.5) frequently have intent deviation ($\geq$ 15.9% average) but fewer logic inconsistencies ($\leq$4.5%). This pattern likely reflects the difference between specialized and general-purpose pretraining. Despite fine-tuning, general models retain broad task knowledge from pretraining, which can lead them to apply reasoning patterns from unrelated tasks—resulting in higher intent deviation.

5 How well do existing metrics detect hallucinations in CodeChange2NL tasks?

RQ1 showed that models often exhibit hallucinations and misinterpretations of code changes. In RQ2, we examine how effective automated approaches are at detecting these hallucinations in code review and commit message generation. Using our manually annotated dataset, we evaluate both reference-based and reference-free metrics described in Section 3.4. Our goal is to assess how well existing metrics detect hallucinations in Code-to-NL tasks, particularly for code changes. We evaluate both individual metrics and combinations of complementary ones to determine whether they can approximate human judgment.

We use ROC-AUC to evaluate the hallucination detection capability of each metric. The positive class is the hallucination samples that we annotated. The negative class is the non-Hallucination samples. A ROC-AUC score of 1 indicates perfect discrimination between hallucinated and non-hallucinated cases, while a score of 0.5 suggests no discriminatory power equivalent to random guessing. For individual metrics, we calculate the ROC-AUC to assess discrimination power.[12] To combine metrics, we use logistic regression and evaluate its performance using accuracy and ROC-AUC.

5.1 How do individual metrics perform in detecting hallucinations?

Metric Effectiveness. Based on the the generator-agnostic results, the current metrics achieve modest ROC-AUC scores ranging from 0.538–0.566 on CodeReviewer and 0.562–0.617 on CommitBench (see Figures 2 and 3). Based on the generator-specific results, hallucinations in CCT5 are more detectable on the CodeReviewer dataset (ROC-AUC 0.65-0.71), while hallucinations in Llama3.1 are most detectable on the CommitBench dataset (ROC-AUC 0.62-0.68). This suggests that the effectiveness on hallucination detection of the metrics may vary across generation models and datasets.

Table 4 shows the metrics with the highest ROC-AUC scores in each studied dataset. In addition, we observe that on CodeReviewer, uncertainty-based metrics (logit and entropy) perform best, while embedding similarity and reference-based metrics are best on CommitBench. Nonetheless, the ROC-AUC scores suggest the limited effectiveness of current metrics on hallucination detection, which are slightly better than random guessing, highlighting the challenges of automated hallucination detection in these tasks.

Metric Complementarity. Different metrics may capture distinct aspects of hallucinations, potentially flagging different instances. To assess this, we selected the three highest-performing metrics based on ROC-AUC and examined their top 25% ranked samples (see the analysis details in Appendix D.2). Figure 4 shows small overlap in the

[10] The full code context in provided in Appendix C.1.

[11] The code patch is provided in Appendix C.2.

[12] The point-biserial correlation confirms a similar trend between metric scores and hallucination labels. Detailed results are provided in Appendix D.3.

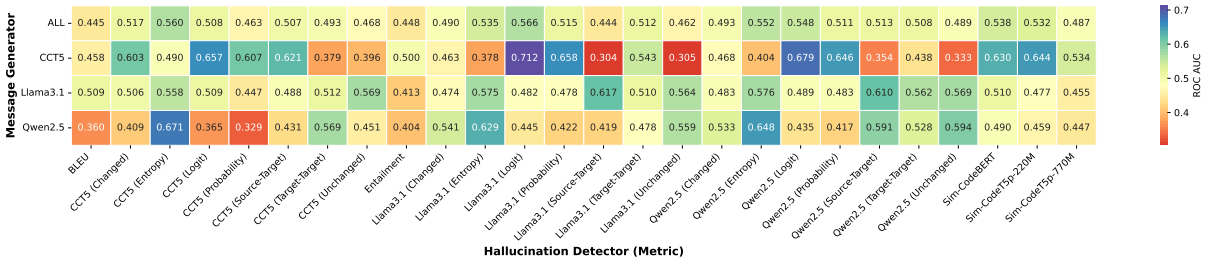


Figure 2: ROC-AUC Scores of Metrics for Hallucination Detection Across Generators on CodeReviewer. The ALL row represents the generator-agnostic result, using all outputs from CCT5, Llama3.1, and Qwen2.5. The remaining rows show performance in the generator-specific result, based on outputs from each model individually.

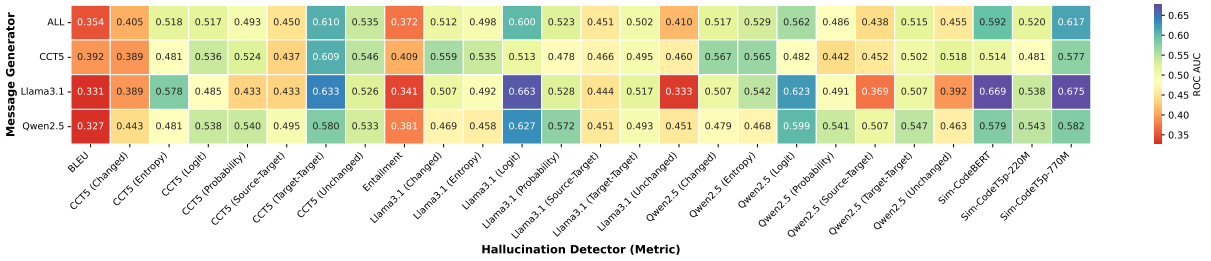


Figure 3: ROC-AUC Scores of Metrics for Hallucination Detection Across Generators on CommitBench.

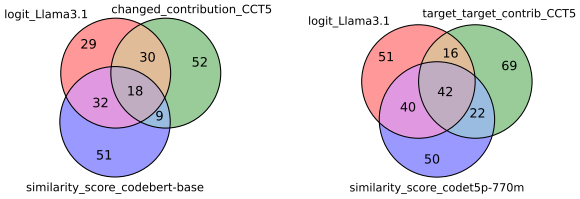


Figure 4: Top 3 individual metrics complement to each other on CodeReviewer (left) and CommitBench (right)

	CodeReviewer		CommitBench	
Type	**Acc**	**AUC**	**Acc**	**AUC**
Top Performing Individual Metrics				
logit_Llama3.1	-	0.57	-	0.60
Sim-CodeT5p-770M	-	0.48	-	0.62
Sim-Codebase	-	0.54	-	0.59
changed_contrib_CCT5	-	0.52	-	0.41
target_target_contrib_CCT5	-	0.49	-	0.61
Multiple Metrics on Logistic Regression				
Reference-based	81.6	0.59	76.0	0.68
Reference-free	81.6	0.66	78.9	0.75
ALL	**82.7**	**0.69**	**77.8**	**0.75**

Table 4: Logic regression results (Acc (%) and AUC) on hallucination prediction using multiple metrics.

top 25% samples ranked by these three metrics, indicating these metrics flag different instances as hallucinated. This highlights the potential complementarity between metrics.

5.2 Can combining multiple metrics enhance the accuracy of hallucination detection?

The results in section 5.1 highlight the potential complementarity between metrics. Thus, we explore whether combining them can improve performance. Prior work (Snyder et al., 2024) also shows that combining multiple signals improves hallucination detection in question-answering tasks. To analyze the descrimination power of combined metrics for hallucination detection, we use a logistic regression model fitted to our annotated samples. For each generation task, we combine all samples from the three models, resulting in 440 samples for CodeReviewer and 717 samples for CommitBench.

To understand the capability of different types of metrics, we build three logistic regression models using: 1) all metrics, 2) reference-based metrics only, and 3) reference-free metrics only. Since some metrics may capture similar signals or redundant, leading to multicollinearity and overfitting, we use the Akaike Information Criterion (AIC) (Akaike, 1974) to identify metrics that meaningfully contribute to the prediction. Then, we use the selected metrics as features to fit the logistic regression model and analyze the coefficients to identify which metrics are most important for hallucination detection.

Table 4 shows the logistic regression results. Combining multiple metrics substantially improves ROC-AUC scores for hallucination detection on both datasets, compared to individual metrics alone.

Type	Metric	|Coef|	Sign
Uncertainty	logit_Llama3.1	6.00*	+
Uncertainty	entropy_Qwen2.5	3.33*	+
Attribution	source_target_Qwen2.5	2.83*	+
Attribution	source_target_Llama3.1	2.78*	-
N-gram	BLEU	1.94*	-

Table 5: Top-5 important features on predicting hallucinations (vs. non-hallucinations) in CodeReviewer. $*$ indicates the coef is significant ($p < 0.05$).

Type	Metric	|Coef|	Sign
Uncertainty	logit_Llama3.1	6.86*	+
Uncertainty	logit_Qwen2.5	5.93*	-
Attribution	changed_CCT5	4.71*	-
N-gram	BLEU	3.49*	-
Similarity	similarity_score_codebert	2.41*	+

Table 6: Top-5 importance features on predicting hallucinations (vs. non-hallucinations) in CommitBench.

For CodeReviewer, the ROC-AUC increased from the best individual score of 0.57 (logit_Llama3.1) to 0.69 when using all metrics. For CommitBench, it improved from 0.62 (similarity_score_codet5p-770m) to 0.75. Surprisingly, using reference-free metrics alone achieved ROC-AUC scores close to that of using all metrics. In contrast, reference-based metrics achieved lower performance, possibly because they are fewer in number or inherently less predictive. This highlights a potential benefit of hallucination detections in these `CodeChange2NL` tasks without ground-truth.

Tables 5 and 6 present the most important features along with their coefficients. The SeqLogit calculated with Llama3.1 (Logit_Llama3.1) emerges as the most important feature for both tasks. Uncertainty metrics from Llama3.1 and Qwen2.5 consistently appear among the top features, demonstrating strong predictive power.For CommitBench dataset, the $-$ coefficient in Qwen2.5 aligns with prior findings, i.e., when not hallucinating, a model is more confident (Dale et al., 2023). On the other hand, the $+$ coefficient in Llama3.1 could be due to its overconfidence as the distribution of logit in Llama3.1 is skewed towards high scores in CommitBench (See Appendix D.4). Feature attribution metrics rank next in predictive strength, indicating that hallucinations can be detected by analyzing how models utilize source code during generation.

Figure 5 presents an example generated for code review.[13] The generated review suggests passing a parameter that is already being passed in both

[13] See an example of commit message in Appendix Figure 8.

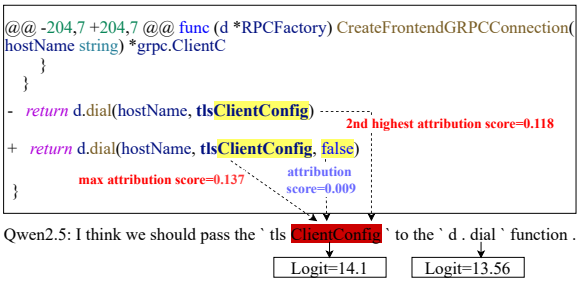


Figure 5: An example of feature attribution on a hallucinated code review comment generated by Qwen2.5. Attribution model: Llama3.1.

old and new code, while ignoring the actual code change. This hallucinated generation has high logit and high attribution from source code. Particularly, the generated tokens appearing in the input context have high confidence based on elevated logit values. For example, based on uncertainty calculated with Llama3.1, particularly API method names like `tlsClientConfig` and `dial` have logit values of 14.1 and 13.6. However, based on the attribution scores, critical changes (i.e., the addition of the "`false`" parameter) that should be the primary focus of the review has minimal contribution to the generation. Instead, these common tokens like `tlsClientConfig` have large attribution scores, meaning that they contributing significantly to the generation.

For non-hallucinations, we observed that the correct input in the code changes contributes significantly to the relevant generation compared to other code snippets (e.g., in the generated comment, "*Why is this needed?*" the "this" token was mainly contributed by the changed line of code "*+ from databricks import koalas as ks*"). This indicates that the balance between the contribution from changed code and unchanged code is one important cause of hallucination in code review tasks.

6 Conclusion

Hallucinations are prevalent in `CodeChange2NL` tasks, occurring in 50% of code reviews and 20% of commit messages. We identify three common types—input/logic inconsistency, and intention violation. Our findings show that individual metrics are insufficient for effective detection, while a multi-metric approach significantly improves performance, particularly combining model confidence and feature attribution.

7 Limitations

While our study advances the understanding of hallucination severity and automatic detection capabilities in CodeChange2NL tasks, several limitations remain.

Dataset Size. Despite using statistically representative samples from the test set, our annotated dataset is relatively small due to the significant effort required for manual annotation. To mitigate this limitation, we analyzed both model-specific and aggregated samples across models to increase effective sample sizes.

Hallucination Granularity. We primarily focused on instance-level (whole sequence) hallucination analysis to establish a foundational understanding of the phenomenon. Our feature attribution analysis showed promise for token-level hallucination detection, revealing cases where generation heavily relied on unchanged code snippets while ignoring critical changes. Future work should explore finer-grained token-level hallucination analysis with appropriate annotations and develop techniques for more precisely identifying hallucinations at different levels of granularity.

Model Recency and Coverage. Due to cost constraints, we excluded commercial models (e.g., GPT-4o, Claude 3.7) from our analysis and focused on the latest open-source language models available at the time of our experiments. However, the landscape is evolving rapidly, with newer models such as LLaMA 4 and Qwen2.5-Coder emerging since our evaluation. As a result, our findings may not fully generalize to these newer or commercial models, or to different model families such as Gemini, which could exhibit different hallucination patterns in Code2NL tasks. Also, our study focuses on the hallucination in task-specific fine-tuned models since they perform better than zero-shot prompting. The hallucination prevalence in zero-shot prompting may be different. Our work lays the foundation for future research in this space, highlighting the need for ongoing evaluation as models continue to evolve and diversify.

References

Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. Codemirage: Hallucinations in code generated by large language models. *arXiv preprint arXiv:2408.08333*.

H. Akaike. 1974. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723.

Berkay Berabi, Alexey Gronskiy, Veselin Raychev, Gishor Sivanrupan, Victor Chibotaru, and Martin Vechev. 2024. Deepcode ai fix: Fixing security vulnerabilities with large language models. *arXiv preprint arXiv:2402.13291*.

Yuyan Chen, Zehao Li, Shuangjie You, Zhengyu Chen, Jingwen Chang, Yi Zhang, Weinan Dai, Qingpei Guo, and Yanghua Xiao. 2025. Attributive reasoning for hallucination diagnosis of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 23660–23668.

David Dale, Elena Voita, Loic Barrault, and Marta R. Costa-jussà. 2023. Detecting and mitigating hallucinations in machine translation: Model internal workings alone do well, sentence similarity Even better. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 36–50, Toronto, Canada. Association for Computational Linguistics.

Mohamed Elaraby, Mengyin Lu, Jacob Dunn, Xueying Zhang, Yu Wang, Shizhu Liu, Pingchuan Tian, Yuping Wang, and Yuxuan Wang. 2023. Halo: Estimation and reduction of hallucinations in open-source weak large language models. *arXiv preprint arXiv:2308.11764*.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems . In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53, Los Alamitos, CA, USA. IEEE Computer Society.

Samuel Ferino, Rashina Hoda, John Grundy, and Christoph Treude. 2025. Junior software developers' perspectives on adopting llms for software engineering: a systematic literature review. *arXiv preprint arXiv:2503.07556*.

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

Luca Di Grazia, Paul Bredl, and Michael Pradel. 2023. Diffsearch: A scalable and precise search engine for code changes. *IEEE Trans. Softw. Eng.*, 49(4):2366–2380.

Nuno M. Guerreiro, Elena Voita, and André Martins. 2023. Looking for a needle in a haystack: A comprehensive study of hallucinations in neural machine translation. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1059–1075, Dubrovnik, Croatia. Association for Computational Linguistics.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

Xiangkun Hu, Dongyu Ru, Lin Qiu, Qipeng Guo, Tianhang Zhang, Yang Xu, Yun Luo, Pengfei Liu, Yue Zhang, and Zheng Zhang. 2024. Refchecker: Reference-based fine-grained hallucination checker and benchmark for large language models.

Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2).

Yuheng Huang, Jiayang Song, Zhijie Wang, Shengming Zhao, Huaming Chen, Felix Juefei-Xu, and Lei Ma. 2024. Look before you leap: An exploratory study of uncertainty measurement for large language models. In *International Conference on Software Engineering (ICSE)*.

Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1219–1231, New York, NY, USA. Association for Computing Machinery.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.

Sungmin Kang, Louis Milliken, and Shin Yoo. 2024. Identifying inaccurate descriptions in llm-generated code comments via test execution. *Preprint*, arXiv:2406.14836.

Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff is not enough: Generating commit messages leveraging reasoning and action of large language model. *Proceedings of the ACM on Software Engineering*, 1(FSE):745–766.

Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. In *Proceedings of ESEC/FSE*, page 1035–1047.

Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1509–1521, New York, NY, USA. Association for Computing Machinery.

Hong Yi Lin, Patanamon Thongtanunam, Christoph Treude, and Wachiraphan Charoenwet. 2024. Improving automated code reviews: Learning from experience. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 278–283, New York, NY, USA. Association for Computing Machinery.

Chunhua Liu, Hong Yi Lin, and Patanamon Thongtanunam. 2025. Too noisy to learn: Enhancing data quality for code review comment generation. In *Proceedings of the 21st International Conference on Mining Software Repositories*.

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*.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018a. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384.

Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018b. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 373–384, New York, NY, USA. Association for Computing Machinery.

Junyi Lu, Lili Jiang, Xiaojia Li, Jianbing Fang, Fengjun Zhang, Li Yang, and Chun Zuo. 2025. Towards practical defect-focused automated code review. In *Forty-second International Conference on Machine Learning*.

Kishan Maharaj, Vitobha Munigala, Srikanth G Tamilselvam, Prince Kumar, Sayandeep Sen, Palani Kodeswaran, Abhijit Mishra, and Pushpak Bhattacharyya. 2024. Etf: An entity tracing framework for hallucination detection in code summaries. *arXiv preprint arXiv:2410.14748*.

Potsawee Manakul, Adian Liusie, and Mark Gales. 2023. SelfCheckGPT: Zero-resource black-box hallucination detection for generative large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9004–9017, Singapore. Association for Computational Linguistics.

Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919, Online. Association for Computational Linguistics.

Sabrina J. Mielke, Arthur Szlam, Emily Dinan, and Y-Lan Boureau. 2022. Reducing conversational agents'

overconfidence through linguistic calibration. *Transactions of the Association for Computational Linguistics*, 10:857–872.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of ACL*, pages 311–318.

Gabriele Sarti, Nils Feldhus, Ludwig Sickert, and Oskar van der Wal. 2023. Inseq: An interpretability toolkit for sequence generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 421–435, Toronto, Canada. Association for Computational Linguistics.

Maximilian Schall, Tamara Czinczoll, and Gerard De Melo. 2024. Commitbench: A benchmark for commit message generation. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 728–739, Potsdam, Germany. IEEE.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3145–3153. PMLR.

Ben Snyder, Marius Moisescu, and Muhammad Bilal Zafar. 2024. On early detection of hallucinations in factual question answering. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 2721–2732, New York, NY, USA. Association for Computing Machinery.

Joël Tang, Marina Fomicheva, and Lucia Specia. 2022. Reducing hallucinations in neural machine translation with feature attribution. *arXiv preprint arXiv:2211.09878*.

Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. 2012. How do software engineers understand code changes? an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA. Association for Computing Machinery.

Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, pages 2389–2401.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2024. Codehalu: Investigating code hallucinations in llms via execution-based verification. *arXiv preprint arXiv:2405.00253*.

Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards automating code review activities. In *Proceedings of ICSE*, pages 163–174.

Simon Valentin, Jinmiao Fu, Gianluca Detommaso, Shaoyuan Xu, Giovanni Zappella, and Bryan Wang. 2024. Cost-effective hallucination detection for llms. In *KDD 2024 GenAI Evaluation Workshop*.

Lanxin Yang, Jinwei Xu, Yifan Zhang, He Zhang, and Alberto Bacchelli. 2023. Evacrc: Evaluating code review comments. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 275–287, New York, NY, USA. Association for Computing Machinery.

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Yichi Zhang. 2024. Detecting code comment inconsistencies using llm and program analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 683–685, New York, NY, USA. Association for Computing Machinery.

Kaitlyn Zhou, Dan Jurafsky, and Tatsunori Hashimoto. 2023. Navigating the grey area: How expressions of uncertainty and overconfidence affect language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5506–5524, Singapore. Association for Computational Linguistics.

A Hallucination Annotation

We used the annotation workflow described in Section 3.2 to guide the process of identifying and labeling hallucinations. Detailed definitions for each node (both non-hallucination and hallucination classes) are provided in Table 7.

To help annotators understand the essential elements of commit messages and code review comments, task definitions were also provided in A.1. Through initial pilot rounds and discussions among annotators, we distilled a set of rules to guide the annotation process, which is provided in A.2.

Type: Definition
Semantic Equivalent (SE): The generated message is semantically equivalent to the ground truth. • In code review, a semantically equivalent comment should share the same intentions regarding both the issues identified and the solutions proposed as in the ground truth. • In commit message, we should consider both the "What" and "Why" together to decide the semantic equivalence. Semantic equivalent commit messages should convey the same intents with similar framing and emphasis.
Not_SE_Informative: M is different from ground truth but it is informative for the task as hand. • In code review, M is considered as informative if it points out a concern and/or provide suggestions for improvement. • For commit messages, M captures some aspects of the code change but may overlook certain points compared to the ground truth. For instance, *'Add 'scheme' to sys path in ok_test/scheme.py"* indicates where the change occurs but lacks the 'why.' In contrast, the ground-truth message *Add 'scheme' to path to handle zip archive case"* provides (why) context on the purpose of the modification. Note (simple way): M must contain "What", but can be incomplete or slightly different from ground truth; "Why" can be missing.
Not_SE_Uninformative: M is different from the ground truth and it doesn't provide useful information for the task at hand. • In code review, M is considered uninformative if it merely seeks information to understand the code design or implementation choices, presents a general question without rationale, serves as self-justification for the code change, or acts as a compliment to the code. Note (simple way): if the What (issue) is missing, then it's not informative. • In commit messages, vague and general wording fails to clearly communicate the specifics of the change, such as the 'what' (the nature of the modification) and the 'why' (the reason for the modification). For example, the message *'Minor refactoring in VRaptor'* lacks detail about what parts were refactored and the intended impact of those changes, making it difficult for reviewers to understand the significance or context of the update. Note (simple way): "What" is essential, it's uninformative if it lacks specifics of "What".
Unsure_or_Looks_Applicable: M appears relevant to the context but needs further fact-checking, as its factual accuracy cannot be directly verified from the given context • In code review, this can involve M using context such as historical background, rationale beyond the given input, or the need for fact-checking the provided solution. • In a commit message, the rationale for explaining the issue or objectives in M might need fact-checking.
Input Inconsistency : M conflicts with the provided input. • In code review, this means M points out an non-existent issue or provides a solution that is already exists in the code change or violates with programming commonsense. • In commit message, this means that M contains information that's not included in the code change, or misinterpret code change.
Logic Inconsistency: M itself doesn't make logical sense.
Context Repetition: M is completely or largely copied from the input.
Intent Deviation: M deviates with the goal of the task at hand: not providing a review in code review task or not providing a commit message that covers what is being changed and why it's being changed.
Others: This is used to capture any other types that's not covered in the above categories

Table 7: The definitions for each of the type in our annotation. M denotes the model generated message.

A.1 Essential Elements in Code Reviews and Commit Messages

Code Review Comments The primary purpose of code review comments is to offer constructive feedback from reviewers to code authors, aiming to improve code quality and maintain coding standards. A review comment often covers three elements:

- What (Evaluation): A review comment should point out what is the concern or issue in the code (Yang et al., 2023).
- How (Suggestion): An ideal review comment provides suggestions for correction or prevention since code review is expected to help fix defects, improve quality, and address developers' quality concerns (Yang et al., 2023).
- Why: Explain the reasoning behind the concern and/or the suggested improvement (Lin et al., 2024).

Commit Messages The primary purpose of commit messages is to provide developers (both current and future) with a summary of code changes, enabling them to understand how the code of a project has changed and why. Two elements have been shown to be essential for a commit message (Liu et al., 2018b; Tian et al., 2022).

- What (Changes): A summary of what changes were made in the code. This often includes:
  - A summary of code object change that shows the object of change, characteristics of changes, or contrast before and after. For example, *"this commit removes the following deprecated prop-*

*erties: * 'server.connection-timeout' * 'server.use-forward-headers' [...]"*. Another example, *"rename HeldCertificate.Builder.issuedBy() to signedBy()"*.

  - An illustration of function. For example, *Rename preferred-mapper property so its clear it only applies to JSON*)
  - Description of implementation principles. For example, *"SslContextBuilder was using InetAddress.getByName(null) [...] On Android, null returns IPv6 loopback, which has the name 'ip6-localhost' "*

- Why: A justification of the motivation behind the code change. This often includes describing objectives or issues, illustrating requirements, or implying necessity.

A.2 Summarized rules for annotation

Rules for Annotating Generated Code Reviews

1. Unsure → Knowledge_Overreach: a note of Knowledge_Overreach should be left for cases that contain code snippets or software evolution (maintains, process related), we are not sure whether the generated content is true or not. E.g., *"I think it would be better to use 'getById' here."*

2. For a composite review that contains multiple sentences, there might be some sentences not functioning as review. As long as there is at least one review exist, we consider it as review (not intent deviation).

3. A review might have multiple sentences and each sentence has different labels, we decide the final label based on most severe one (label hallucination types if it exists).
   For example, given this message *"I think this is a bug. The 'm_indirectKernelMem' is a 'std::vector<usm::memory>'. The 'usm_mem' is a single element of that vector. So this line is going to overwrite the 'm_indirectKernelMem' with a single element."*. We have two labels: (a) we cannot tell that the *m_indirectKernelMem' is a 'std::vector<usm::memory>* or not, which is 'Unsure' requires fact checking; and (b) we know that *"So this line is going to overwrite the 'm_indirectKernelMem' with a single element."* is wrong based on the code context, it won't overwrite, so it's Input Inconsistency. Base on the two labels, we choose Input Inconsistency for this message.

4. How to distinguish it's a review or a justification? A review should contain the basic components of issue/concern, with optional suggestion and explanation, while a justification is a message aligned with the code change (no concern or suggestion, no new information inside). For example, this message "This is a bit of a hack, but I think it's the best we can do for now" should be labeled as Intent Deviation since there is no any issue or concern.

5. Cases where the model suggests changing back to the older version without explanation, we don't know whether the suggestion is better or not. If know exactly what to fact check, we label it Unsure (needs fact checking); otherwise, if it's not violating the context, then we choose NO context deviation and then decide whether it's Informative or Uninformative. The following message should be labeled as Context Deviation → No and Informative, because it's sensible given the code context: "I think this is a bit of a misnomer. I think it should be "Gets or sets JSON serialization settings".".

6. In cases where the review is ambiguous, it might refer back to multiple places in the code patch, we label it as No-context deviation if it's possible to apply in at least one kinds of scenario. Leave a comment of "Can be interpreted as another wrong way". In the example of: "Layout/EmptyLinesAroundBlockBody: Extra empty line detected at block body end.", where the 'block body end' can be mapped to different places, one with an extra empty line and one without.

7. A review can apply to multiple places in the code patch, we prioritize mapping it to the code change part (-/+ lines) unless the review explicitly mentions other unchanged code snippets. For example, in this message *"I think this is a bit of overkill. We can just use 'Fatal' and 'Warning' directly."*, the 'Fatal' and 'Warning' exist in both code changed parts and unchanged parts, but we prioritize the changed part.

Rules for Annotating Commit Messages

1. A message is considered as semantically equivalent to the ground truth message if the information you can get are equal after reading both. Specifically, both "what" changed in the code and and "why" it is changed should be aligned.

2. For semantic equivalence, we don't not over-infer the meanings, if the message doesn't explicit mention about it then it's not. E.g., *"Added support for CircleMarker"* we don't infer the CircleMarker is a type/instance of Marker unless the code explicitly defined it.

3. For cases where we are not sure and cannot understand the message based on the given context, our prior knowledge and external web search, label it as Unsure, leave a note of "Difficult to comprehend the message".

4. The <I> symbol comes from training data, where they mask out information referring to a different platform such as issue IDS, URLs, and version numbers. For example, the message *"Bump to <I> (#<I>)"* is not hallucinating, but it's Uninformative based on the code change as it doesn't tell specifics of what bump to <I>. This message *"removed unused imports from rfc<I>"* is considered informative based on the code context.

A.3 red

Examples with Annotated Hallucination Types We use the code review task as a representative example to illustrate the distinctions between different types of hallucinations.

Input Inconsistency We annotate the model output code review *"I think we should use* `Trim()` *here as well."* as an Input Inconsistency, since the corresponding code change (see below) already uses Trim(). While the comment may seem reasonable on its own, it conflicts with the actual code context and addresses a non-existent issue.

```
 @@ -144,6 +144,12 @@ namespace OpenTelemetry.
      Instrumentation.AspNetCore.Implementation
     {
         activity.SetTag(SemanticConventions.
             AttributeHttpUserAgent, userAgent);
     }
+
+             var xForwardedFor = request.
    Headers["X-Forwarded-For"].FirstOrDefault();
+             if (!string.IsNullOrEmpty(
    xForwardedFor))
+             {
+                 activity.SetTag(
    SemanticConventions.AttributeHttpClientIP,
    xForwardedFor.Split(',').First().Trim());
+             }
         }
     }
```

Logic Inconsistency We annotate the following model output as Logic Inconsistency: *"I think it would be better to test that* `` `resp.json['data']['data']['fingerprint']` `` *is a* `` `dict` `` *and not a* `` `dict` ``*."*. The logic inside the review comment itself is problematic, regardless of the code input.

Input Repetition For the code change below, the model output *'input[form] polyfill, cannot load conditionally'* is annotated as *Input Repetition*, as the output merely copies snippets from the code change.

```
@@ -15,6 +15,7 @@ return array(
         'vendor/bootstrap.min.js',
         'vendor/bootstrap-accessibility.min.js',
         'vendor/validator.min.js',
+        'vendor/form-attr-polyfill.js', // input
    [form] polyfill, cannot load conditionally,
    since we need all versions of IE
         'autocomplete.js',
         'common.js',
         'lightbox.js',
```

Intent Deviation

For the code change below, the model output *"This is a bug fix. The* `` `period.fraction` `` *was being used instead of* `` `period.nanosec` `` *in the* `` `DeadlineQosPolicy` `` *constructor."* is annotated as *Intent Deviation*, as it describes the code change rather than providing a review comment. It fails to identify issues or offer suggestions, thus violating the task objective.

```
   @@ -42,7 +42,7 @@ bool DeadlineQosPolicy::
       addToCDRMessage(CDRMessage_t* msg)
    bool valid = CDRMessage::addUInt16(msg, this
        ->Pid);
    valid &= CDRMessage::addUInt16(msg, this->
        length);//this->length);
    valid &= CDRMessage::addInt32(msg,period.
        seconds);
-   valid &= CDRMessage::addUInt32(msg,period.
    fraction);
+   valid &= CDRMessage::addUInt32(msg,period.
    nanosec);
    return valid;
 }
```

B Prompting and Fine-tuning Models

Zero-shot prompting We use vLLM[14] for zero-shot prompting. The model temperature was set

[14] https://docs.vllm.ai/en/latest/

to 0 to make the output deterministic. We used the following prompts for code review and commit message generation.

> Below is a code diff submitted during a code review process.
> Please write a commit message within 50 words.
> `[code_diff]`: `{code_diff}`
> Respond only with valid JSON. Do not write an introduction or summary.

> Below is a code diff submitted during a code review process. Please write a code review comment within 50 words to identify the concerns and suggest improvements.
> `[code_diff]`: `{code_diff}`
> Respond only with valid JSON. Do not write an introduction or summary.

Fine-tuning models We fine-tuned the three models on task-specific training data, including two general language models (Llama3.1-8B-Instruct[15] and Qwen2.5-7B-Instruct[16]) and one specialized small language model pre-trained on code and commit message generation (Lin et al., 2023). The experiment was conducted on 1 NVIDIA H100 GPU.

For CCT5 (Lin et al., 2023), we reused the code and original scripts from their replication package[17] to fine-tune the model on our dataset. The hyperparameters are: train_batch_size= 32, learning_rate = 3e-4, max_source_length = 512, max_target_length = 128 and warmup_steps = 500, gradient_accumulation_steps = 4, maximum_train_steps = 150000, optimizer=AdamW.

For LLaMA3.1-8B-Instruct and Qwen2.5-7B-Instruct, we perform instruction fine-tuning to further update the models parameters for the tasks at hand. We use full fine-tuning rather than parameter-efficient methods such as LoRA, as our preliminary experiments found that full fine-tuning performed better. The following instruction templates are used during training:

> Below is an instruction that describes a task, paired with an input that provides further context. Write an Output that appropriately completes the request.
> ### Instruction: Review the code diff and provide a constructive comment highlighting any issues and suggesting improvements.
> ### Input:
> Code diff: `{code_diff}`
> ### Output:
> `{code_review}`

> Below is an instruction that describes a task, paired with an input that provides further context. Write an Output that appropriately completes the request.
> ### Instruction: You are a programmer who makes the below code changes. Please write a commit message for the below code diff
> ### Input:
> Code diff: `{code_diff}`
> ### Output:
> `{commit_message}`

Regarding the hyperparameters used to fine-tune the two LLMs (Llama3.1-8B-Instruct and Qwen2.5-7B), we set the learning_rate = 5e-5, max_sequence_length = 1024, batch_size = 4. We set the max_steps of fine-tuning to be 30000 and choose the best performing model on the validation set. The optimiser is Adamw.

Results We evaluated seven models in total, including four zero-shot and 2 fine-tuned models,[18] on their capability of generating task-specific messages using the traditional BLEU-4 metric (Papineni et al., 2002). Table 8 presents the experimental results on code review comment generation and commit message generation across prompting and fine-tuning approaches.

The experimental results reveal several key patterns. First, zero-shot prompting approaches consistently underperform fine-tuned models, with BLEU scores ranging from 3.88-4.70% for code review and 8.62-9.72% for commit messages. In contrast, fine-tuned models achieve substantially higher performance, with the specialized CCT5 model reaching 5.58% on code review and 17.45% on commit messages. This highlights the neces-

[15] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

[16] https://huggingface.co/Qwen/Qwen2.5-7B-Instruct

[17] https://github.com/Ringbo/CCT5

[18] We consider the fine-tuned LLMs as different models from the ones before fine-tuning, as their weights have been updated for the tasks.

Setting	Model	CodeReview		CommitBench	
		Overall	Sample	Overall	Sample
Zero-shot prompt	Llama3.1-8B-Instruct	4.22	3.28	9.21	8.89
	Qwen2.5-7B-Instruct	4.70	4.00	8.99	8.62
	Llama3.1-70B-Instruct	3.88	4.09	9.72	9.88
	Qwen2.5-72B-Instruct	4.29	4.31	8.62	8.06
Fine-tuned	Llama3.1-8B-Instruct	5.28	5.25	15.06	15.29
	Qwen2.5-7B-Instruct	5.43	5.73	15.37	15.57
	CCT5	**5.58**	**6.53**	**17.45**	**17.46**

Table 8: Performance (BLEU-4 measured in %) comparison of different models on CodeReview and CommitBench benchmarks under zero-shot and fine-tune settings.

sity of fine-tuning for generating higher-quality CodeChange2NL messages.

Second, code review proves to be a more challenging task compared to commit message generation, with BLEU scores approximately 2-3 times lower across all model configurations. This is sensible given that code review comments require models to critically analyze and provide constructive feedback on code changes, representing a higher cognitive demand than the descriptive nature of commit messages.

The performance on our manually sampled subset closely mirrors the overall dataset performance, with sample BLEU scores showing similar trends (e.g., CCT5 achieving 6.53% vs 5.58% overall for code review), validating the representativeness of our evaluation approach.

C Examples of Code Changes

C.1 Example from CodeReview

```
@@ -65,6 +65,7 @@ public class
    SmartStorePlugin extends ForcePlugin {
  public static final String LIKE_KEY = "
      likeKey";
  public static final String MATCH_KEY = "
      matchKey";
  public static final String SMART_SQL = "
      smartSql";
+ public static final String ORDER_PATH = "
     orderPath";
  public static final String ORDER = "order";
  public static final String PAGE_SIZE = "
      pageSize";
  public static final String QUERY_TYPE = "
      queryType";
```

Human Code Review: Previously you could only order by the field you were searching by. But for full-text search, you can search across all indexed fields, and it didn't make sense not to have a sorting. For backward compatibility, the javascript code uses indexPath as the orderPath when no orderPath is provided.
CCT5 Code Review: I think this should be orderPath instead of orderPathKey.

C.2 Example from CommitBench

```
diff --git a/nomad/server.go b/nomad/server.
    go
index <HASH>..<HASH> 100644
--- a/nomad/server.go
+++ b/nomad/server.go
@@ -1169,7 +1169,12 @@ func (s *Server)
    setupRaft() error {
     }
    } else if _, err := os.Stat(peersFile);
        err == nil {
     s.logger.Info("found peers.json file,
         recovering Raft configuration...")
-    configuration, err := raft.ReadPeersJSON
    (peersFile)
+    var configuration raft.Configuration
+    if s.config.RaftConfig.ProtocolVersion <
    3 {
+      configuration, err = raft.
    ReadPeersJSON(peersFile)
+    } else {
+      configuration, err = raft.
    ReadConfigJSON(peersFile)
+    }
     if err != nil {
       return fmt.Errorf("recovery failed to
           parse peers.json: %v", err)
     }
```

Human Commit Message: Add support in nomad for supporting raft 3 protocol peers.json ### CCT5 Commit Message: nomad: fix peers.json recovery for protocol version 3

D Hallucination Detection

D.1 Hallucination Detection Methodology Details

We adopt existing hallucination measurement metrics, including reference-based and reference-free hallucination detection approaches to address different practical needs. Reference-based metrics serve as valuable benchmarks during model training and evaluation when gold standards are available, while reference-free methods enable hallucination detection in real-world deployment scenarios where reference texts are typically unavailable.

D.1.1 Reference-based Metrics

In reference-based metrics, hallucination is estimated by the quality of a generation y, which is evaluated by comparing against the reference $\hat{y}$ using certain metrics. The hypothesis is that the lower the quality is, the more likely y it is to be a hallucination. We use two metrics that are widely used for quality estimation: Lexical overlap with BLEU, and Natural Language Inference.

Lexical overlap metrics such as BLEU evaluate the n-gram overlap between the y and $\hat{y}$. This type of metric has been widely used in prior work to evaluate the quality of generated commit messages (Liu et al., 2018a; Li et al., 2024) and review comments (Tufano et al., 2021; Li et al., 2022). Recently, it has also been adapted to study the correlation with hallucinations in natural language generation tasks, such as machine translation (Guerreiro et al., 2023; Dale et al., 2023).

Natural Language Inference (NLI). NLI is a standard NLP task that evaluates the logic relationship between a pair of premise and hypothesis sentences, determining whether it is entailment, contradiction, or neutral, which has been widely used to evaluate the factual consistency (Hu et al., 2024; Valentin et al., 2024) and hallucination detection (Manakul et al., 2023; Elaraby et al., 2023). We use NLI to measure the probability of the reference y entails the the generated NL $\hat{y}$. The intuition is that if the y can be directly inferred from the reference $\hat{y}$, then it is high quality and less likely to hallucinate. We used the best performing model nli-deberta-v3[19] based on the performance on Sentence Transformer [20] to obtain the entailment logit.

D.1.2 Reference-free Metrics

In reference-free measurements, reference is not accessed, only information from the source input or from the model behaviors while generating a sequence is used. We use three types of measurements: similarity-based, uncertainty-based, and feature-attribution based.

Similarity between the generation and the source We estimate semantic similarity between source and generation using cosine similarity $\cos(E_y, E_x)$ between embeddings of generated NL y and source code x. The intuition is that irrelevant generations are less similar and more likely to hallucinate. To obtain the embeddings, we use three models pre-trained on both code and natural language corpora: codebert-base[21], codet5p-220m-bimodal[22], and codet5p-770m[23].

Sequence-level confidence scores A sequence-level confidence score has been used in machine

[19] https://huggingface.co/cross-encoder/nli-deberta-v3-base

[20] https://sbert.net/

[21] https://huggingface.co/microsoft/codebert-base

[22] https://huggingface.co/Salesforce/codet5p-220m-bimodal

[23] https://huggingface.co/Salesforce/codet5p-770m

translation for hallucination detection (Guerreiro et al., 2023; Huang et al., 2024), where it is calculated via aggregating token-level uncertainty into sentence level by taking the average across the sequence. Token-level confidence can be measured in various ways. The intuition is when a model hallucinates, it tends to be less confident. Several metrics have been proposed to estimate the token-level uncertainty, including probability, logit and entropy (Guerreiro et al., 2023; Huang et al., 2024; Valentin et al., 2024).

We also use entropy to measure uncertainty: a more uniform token distribution (higher entropy) indicates lower model certainty. This can be formulated as follows:

$$\text{SeqEntropy} = \frac{1}{L}\sum_{i=1}^{L} H_i, \tag{1}$$

where H_i is the entropy of the token distribution.

Feature attribution In a transformer-based model M, generating a token y_t involves both the input x and previously generated target tokens (y_1 to y_{t-1}). Prior work has shown that the interaction between y_t and these sources reveals hallucination patterns (Tang et al., 2022; Chen et al., 2025; Snyder et al., 2024), which can be detected through feature attribution in NL hallucinations. We conduct both feature attribution for both the input source x and the previously generated target tokens.

We employ a widely used feature attribution method Input X Gradient (Shrikumar et al., 2017), which calculates the gradient of the output with respect to the input and considers the impact of input magnitudes on generation. The attribution score from x_i to y_t can be formulated as:

$$A_{i,t} = x_i \times \frac{\partial y_t}{\partial x_i} \tag{2}$$

where $A_{i,t}$ is the attribution score, and $\frac{\partial y_t}{\partial x_i}$ denotes the gradient of y_t in an attribution model M with respect to the input x_i. A higher $A_{i,t}$ indicates that x_i is more important for generating y_t.

Source Attribution Score. To investigate hallucinations on sequence level, we apply an aggregation function on A to convert a sequence of token-level attribution scores into a single attribution value. We first compute the maximum attribution value across all input tokens for each output token y_t, then take the average of these maximum values. The attribution score of the source to the generated sequence.

$$\text{SourceAttr} = \frac{1}{T}\sum_{t=1}^{T} \max_{i\in[1,N]} A_{i,t}, \tag{3}$$

where T is the length of the generated sequence, SourceAttr represents final sequence-level overall source contribution score. The intuition is that when the maximum input contribution is small, the generated y is likely to be a hallucination as the model didn't generate based on the input.

Given our input is a code change consisting of both old and new code, human developers primarily focus on the changed parts when generating commit messages and code review comments. Based on this observation and the assumption that models should similarly emphasize code changes, we designed variations of the aggregation methods that separate attribution scores for changed and unchanged code. Our hypothesis is that lower attribution scores on the changed parts indicate a higher likelihood of hallucination.

$$\text{ChangedAttr} = \frac{1}{T}\sum_{t=1}^{T} \max_{i\in C} A_{i,t}, \tag{4}$$

$$\text{UnchangedAttr} = \frac{1}{T}\sum_{t=1}^{T} \max_{i\in[1,N]\setminus C} A_{i,t}, \tag{5}$$

where $C \subset [1, N]$ represents the indices of tokens in the changed code (all - and + lines), and $[1, N] \setminus C$ represents the indices of unchanged code tokens.

Target Attribution. We also calculate the attribution score from previously generated tokens:

$$\text{TargetAttr} = \frac{1}{T}\sum_{t=1}^{T} \max_{j\in 1,\ldots,t-1} \hat{A}j, t, \tag{6}$$

where $\hat{A}j, t$ is the attribution score from y_1 to y_j (j ranges from 1 to $t-1$). The final TargetAttr score denotes the overall maximum attribution score from previously generated tokens to the current token.

To obtain attribution scores for generated sequences, we use constrained attribution (Sarti et al., 2023) through the Inseq library.[24] Constrained attribution works by providing an attribution model M with both the input code x and the generated output y, then analyzing how the model associates each input token with each output token step by step. Rather than generating text freely, the model is

[24] https://inseq.org/en/latest/

constrained to follow the specified target sequence, allowing us to measure which parts of the input most strongly influence each token in the output. This reveals the model's implicit justification for each output token based on the input.

As the attribution model M, we use the same three models fine-tuned in our RQ1 experiments for each task: LLaMA3.1-8B-Instruct, Qwen2.5-7B-Instruct, and CCT5. For each generation, we apply both self-attribution (where the generator attributes its own output, e.g., CCT5 attributes its own generation) and cross-attribution (where a different model attributes the output, e.g., CCT5 attributes LLaMA3.1-8B's generation). This dual perspective helps us understand whether a model is aware of its own hallucinations and whether external models can detect hallucinations based on attribution signals. While attributing each output token, we also extract uncertainty scores based on logit, probability, and entropy.

D.2 Complementarity Among Individual Detection Metrics

To examine how different types of metrics complement each other, we select the top three individual metrics (one from each category) based on ROC-AUC.

For CodeReviewer, we choose logit_Llama3.1, similarity_score_codebert-base, and changed_contribution_CCT5. For CommitBench, we select similarity_score_codet5p-770m, target_target_contrib_CCT5, and logit_Llama3.1.

From each metric, we extract the top 25% samples ranked by their metric score, indicating that they are highly correlated with hallucination labels. We then analyze the overlaps and unions of these sets.

Figure 4 shows the Venn diagrams of the selected metrics. On CODEREVIEWER, the three metrics capture almost disjoint sets. On COMMITBENCH, only three samples are shared across all three metrics, suggesting strong complementarity.

D.3 Correlation between Detection Metrics and Hallucination

In addition to ROC-AUC, we also analyzed the correlation between each individual metric and the hallucination labels we annotated (hallucination = 1, non-hallucination = 0). To evaluate the correlation, we use the point-biserial correlation coefficient (r_{pb}), which measures the strength and direction of the relationship between a continuous variable (i.e., metric scores) and a dichotomous variable (i.e., the binary hallucination label).

The results are presented in Figures 6 and 7. Overall, the correlation is weak ($|r_{pb}| \in [0, 0.2)$) across all samples for individual metrics. However, when examining generator-specific results, the correlation between certain generator–metric pairs increases ($|r_{pb}| \in [0.2, 0.3)$).

These findings further motivate our exploration of how combining multiple metrics can improve hallucination detection.

D.4 Signs of Coefficients in LR model

In Section 5.2 (Table 6), we observed that the two uncertainty-based metrics—logit_Llama3.1 and logit_Qwen2.5—both contribute significantly to hallucination prediction, but with opposite coefficient signs: positive for logit_Llama3.1 and negative for logit_Qwen2.5. The signs of the coefficients indicate that higher logits from LLaMA3.1 are associated with hallucinations, whereas higher logits from Qwen2.5 are associated with non-hallucinations. We hypothesize that Qwen's confidence is more reliable, while LLaMA3.1 tends to be overconfident. To further explore this, we plot the joint distribution of the two logits in Figure 9. When Qwen2.5 is more confident than LLaMA3.1 (above the diagonal), hallucinations are less frequent; conversely, when LLaMA3.1 is more confident (below the diagonal), hallucinations occur more often. This pattern supports our hypothesis.

This observation aligns with prior work (Zhou et al., 2023; Mielke et al., 2022), which shows that models can be overconfident when generating outputs due to differences in training data and strategies. In our study, both models were fine-tuned on the same data, so we suspect this difference is partly due to pre-training.

D.5 LR Model Predictions by Hallucination Type

To understand which hallucination types are correctly detected, we examine samples predicted as hallucinations by our best logistic regression models on CodeReviewer and CommitBench (Section 5).

Figure 10 shows the type distributions. They largely mirror the overall dataset distribution, with INPUT INCONSISTENCY most frequent in both datasets, followed by INTENT DEVIATION in CODEREVIEWER, and LOGIC INCONSISTENCY thereafter.

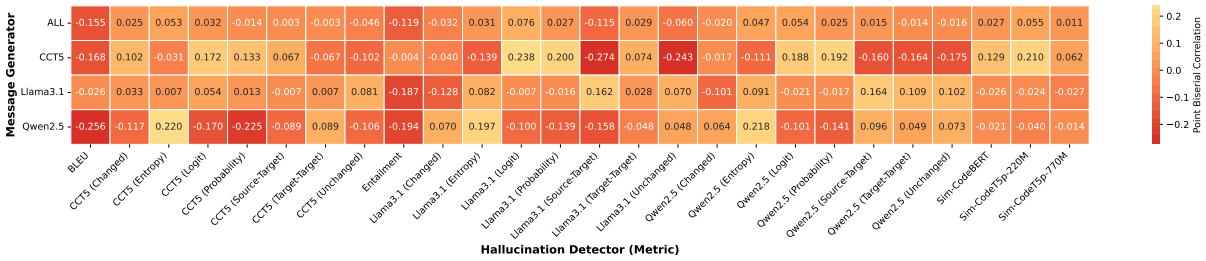


Figure 6: Point-biserial correlation between metrics and hallucinations on CodeReviewer.

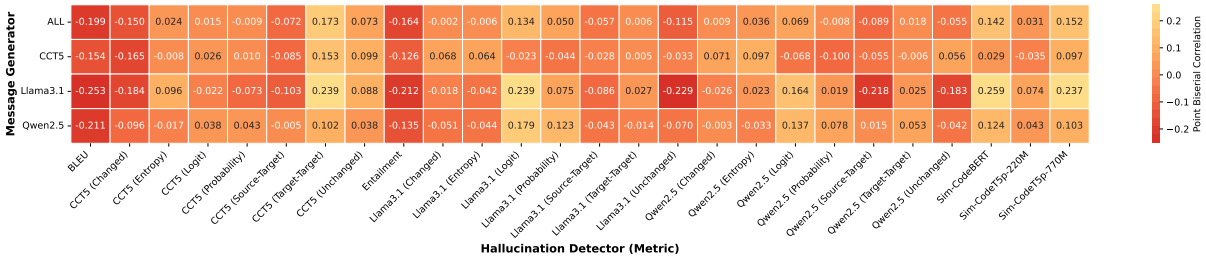


Figure 7: Point-biserial correlation between metrics and hallucinations on CommitBench.

D.6 LR model prediction per programming language

While our hallucination detection approach is language-agnostic, model performance may still be influenced by programming language distributions in pre-training and fine-tuning data. To examine this, we analyze the distribution of programming languages among samples predicted as hallucinations by the logistic regression model and compare it to the distribution of samples labeled as hallucination in the full test set.

The results are shown in Figure 11 for CODEREVIEWER and Figure 12 for COMMITBENCH. In CODEREVIEWER, the language distribution of model predictions closely matches that of the test set, suggesting consistent detection across languages. In COMMITBENCH, the distributions also largely align, with one notable exception: JavaScript (`js`) is the most dominant in the test set but is not predicted (recalled) in the model's predicted hallucinations.

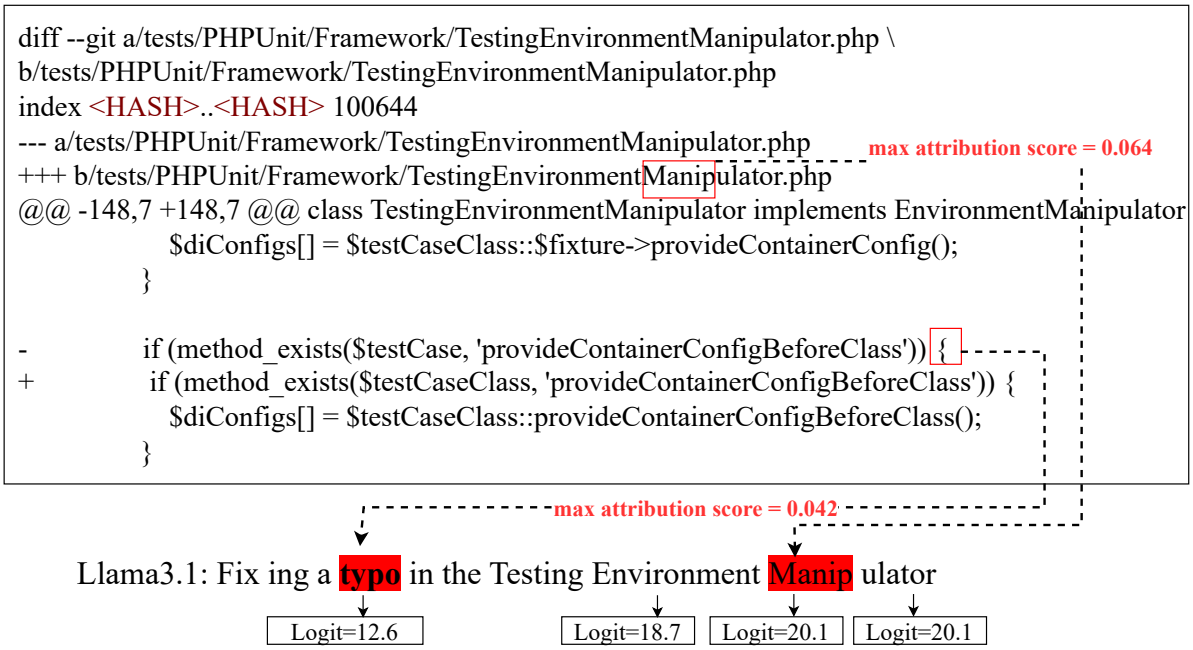


Figure 8: An example of feature attribution on a hallucinated commit message comment generated by Llama3.1. Attribution model: Llama3.1.

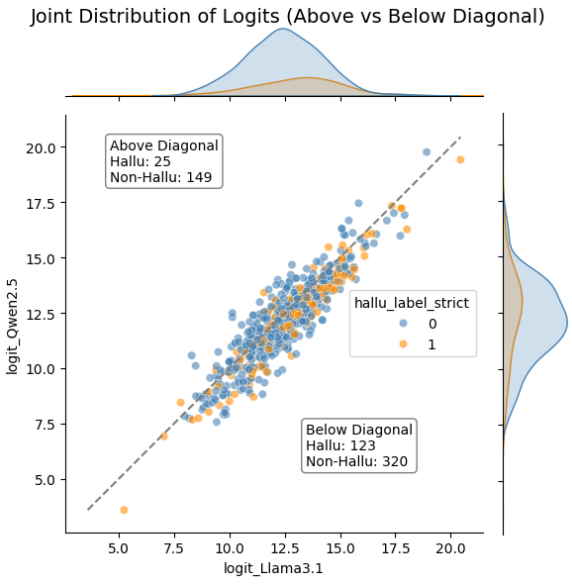
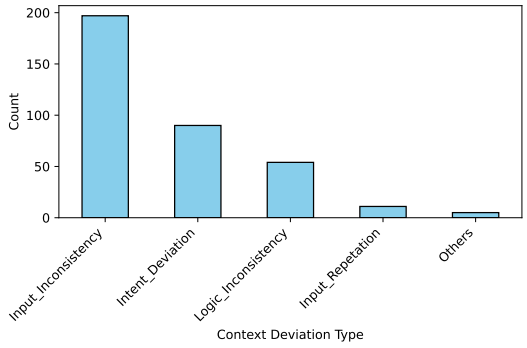
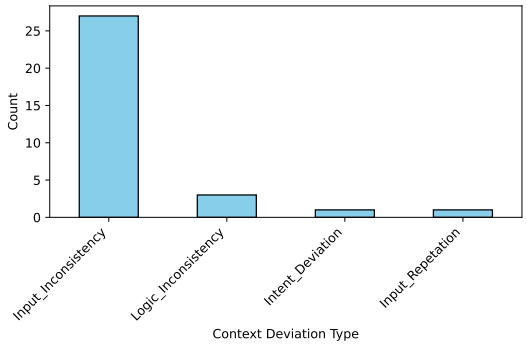


Figure 9: Joint Distribution of Qwen and Llama Logits on CommitBench dataset.

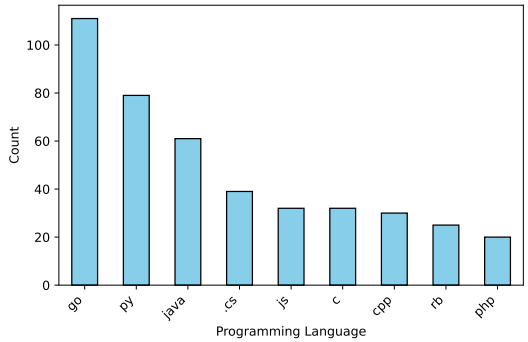


(a) Codereviewer

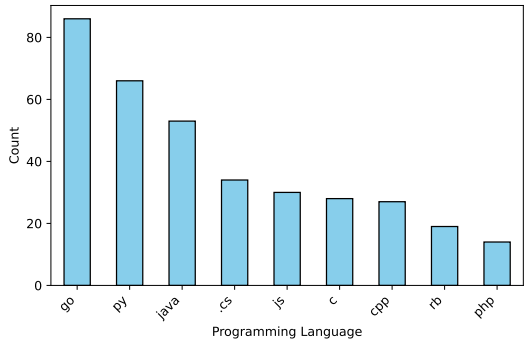


(b) Commitbench

Figure 10: hallucination type distribution on LR models corrected predicted as hallucination.

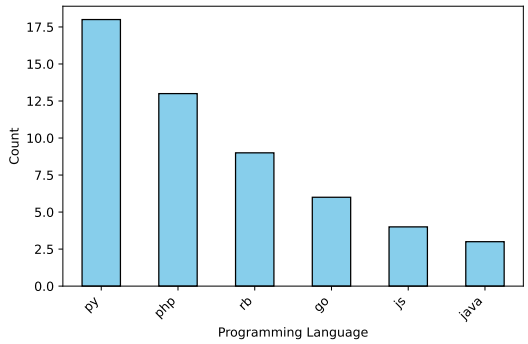


(a) Samples in model corrected predicted as hallucination

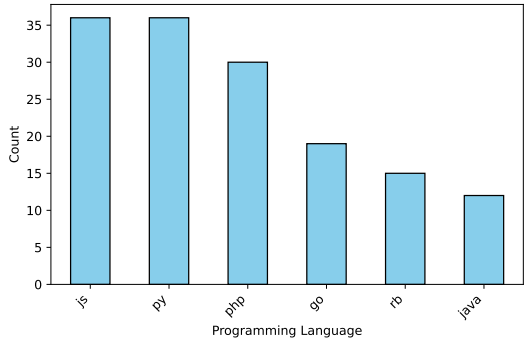


(b) Samples in test set

Figure 11: CodeReviewer: programming language distribution on the model corrected predicted as hallucinations and our test set.



(a) Samples in model corrected predicted as hallucination



(b) Samples in Test set

Figure 12: CommitBench: programming language distribution on the model corrected predicted as hallucinations and our test set.