A Unified Trust Metric System for Detecting Hallucinations in LLM-Generated Code and Summaries

MAX MUSTERMANN, Hochschule München, Germany

Large Language Models (LLMs) have shown that they can make natural language content and source code very well. However, they often make outputs that are hallucinated, ungrounded, or unreliable, which makes it hard for them to be used in the real world. Current methods for detecting hallucinations are mostly broken up and only look at one part at a time, like how true a piece of text is or how well a piece of code runs. They don't give a single, easy-to-understand measure of trustworthiness. In this paper, we propose a Trust Metric System for analyzing the authenticity of AI/LLM-generated responses by integrating Retrieval-Augmented Generation (RAG) with a multi-modal validation framework. The proposed system performs post-generation verification across multiple modalities, including token-level code analysis, execution-based validation, entity-level summary verification, and API-usage correctness. Features extracted from these heterogeneous validation pipelines are fused using a hybrid CNN–MLP architecture, enabling the model to capture both sequential and non-sequential patterns indicative of hallucinations.The system creates a single, easy-to-understand Trust Score by combining its predictions about how likely the generated code and its natural language summary are to have issues. This score helps users make better decisions regarding AI outputs, giving them a clear and measurable way to assess reliability. The proposed framework aims to encourage safer use of large language models in software development and other critical areas, boost user confidence, and enhance accountability. The goal of this experimental evaluation is to show how effective this unified trust metric is by checking both its accuracy in parts and how well it aligns with expert judgment

1 INTRODUCTION

As a result of the widespread introduction of LLM's (Large Language Models) into Software Engineering and other related fields, it was possible for LLM-based tools to create Automated Code Generation, Automatic Code Summarization, and support for Natural Language Programming. The introduction of LLM's into Software Engineering generated considerable improvement in Developer Productivity, enabling Software Engineers to produce functional source code with less time invested and fewer resources required to complete the task. However, although we have seen improvements in Developer Productivity from LLM-based Tools and how to create automated source code, there is a continued difficulty of discovering or determining whether or not an output of an LLM Tool is reliable. This is often referred to as Hallucinations that result from creating outputs for the user without having a valid representation of the developer's intent or actual factual information to produce the correct result in the execution of the tool. The most common examples of Hallucination in software include collecting Intent-Conflicting Logic, Contextually Inconsistent Outputs, Non-Functional or Dead Code, and Improper APIs (Application Programming Interfaces); All of these forms of hallucinations can negatively impact Software Quality (Correctness), Software Maintainability, and the Security of Software In contrast to Hallucinations found in Natural Language, Hallucinations found in code only become apparent when running the code or deploying the code, making it highly hazardous for real-world software systems. Although LLM's eventually become widespread in development workflows, currently, there is no definitive way to assess the reliability of LLM-generated code, limiting safe adoption of LLM's within Developer Workflow. Several research initiatives have sought to address this issue by analysing the various perspectives of Hallucinations.Numerous taxonomies and benchmarks have been created in previous research studies that classify the patterns of hallucination that occur when generating code using LLMs, and it has been proven that the top-tier LLMs consistently fail at recognizing and mitigating these errors.2368.16577 Therefore, these findings give us great insight into the distribution and qualities associated with hallucinations, but, as diagnostic tools, they do not provide a complete end-to-end user-metric for quantifying the trustworthiness of LLM-generated code. 2275.04813 There are ongoing research efforts to detect hallucinations when summarizing code using entity level validation approaches (ex: Entity Tracing Framework) and it shows that even with relatively simple code, the generated summaries are often incorrect with respect to the identified entities or have misrepresented intent. 2455.07883 EtF provides for the interpretation and fine-level detection of hallucinations, but it is limited to validating summaries only; thus it cannot facilitate the assessment of holistic case analysis around the execution of code, the correctness of logic used in the computation, or how the API is intended to be used. 2267.07717 A subsequent area of research is to promote the active use of interactive mitigations (e.g. test-driven development workflows) to allow for increased user engagement in generating code that is produced with large language models. This research indicates that with user participation in developing code through the framework of test-driven development, the probability of producing correct code is greatly enhanced, while simultaneously reducing cognitive demand. The aforementioned methods rest heavily on continuous human feedback to operate, which restricts their potential for scalability and applicability in scenarios where fully automated or real-time confirmation is warranted.

Still, despite recent progress, there exists one critical research gap: A cohesive, automated framework for testing how trustworthy LLMs are across multiple dimensions and providing the results as a single interpretive measure. Currently, each of the current methods have been developed independently of one another, with none providing an aggregated measure of how reliable an LLM's answer is relative to the three aspects listed earlier, namely code execution, summary faithfulness and user feedback.

Our solution is the Trust Metric System, which allows users to validate LLM-generated responses in real-time and bridge the research gap noted above. We introduced a combination of Retrieval-Augmented Generation (RAG) architecture with a multimodal post-code generation verification system. In the proposed model, code is checked for correctness; summaries are compared for faithfulness; and the use of APIs is confirmed. The system integrates token level

Author's Contact Information: Max Mustermann, mustermann@hm.edu, Hochschule München, Germany.

analysis of code; execution results; entity level certification of summaries; and API verification within a hybrid learning framework to produce a single quantifiable Trust Score that can be used to provide an accurate real-time indication of the dependability of outputs from AI systems and to reduce the risk associated with shifting LLMs into high-risk software development environments.

2 LITERATURE SURVEY

Table 1. Literature Survey on Hallucination Detection in LLM-Generated Code

Sl. No	Paper Title	Key Contribution and Research Gap
1	Exploring and Evaluating Hallucinations in LLM-Powered Code Generation	Proposed the HALLUCODE benchmark and taxonomy for hallucinations in LLM-generated code. Focuses on analysis and classification but lacks a deployable trust evaluation metric.
2	ETF: An Entity Tracing Framework for Hallucination Detection in Code Summaries	Introduced entity-level tracing for summary validation. Limited to summaries only and does not validate executable code or API usage.
3	COLLU-BENCH: A Benchmark for Predicting LLM Hallucinations in Code	Used token-level probabilities and execution feedback for hallucination prediction. Does not provide a user-facing trust metric.
4	On Mitigating Code LLM Hallucinations with API Documentation	Introduced CloudAPIBench to reduce API hallucinations. Lacks holistic post-generation verification and trust aggregation.
5	CodeHalu: Investigating Code Hallucinations via Execution-based Verification	Proposed execution-based verification. Fails to capture semantic, summary, and API-level hallucinations.
6	CodeMirage: Hallucinations in Code Generated by LLMs	Introduced a taxonomy and benchmark dataset using prompt-based detection. Lacks an integrated trust evaluation framework.
7	LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation	Studied repository-level hallucinations and proposed RAG-based mitigation without post-generation validation.
8	LLM-Based Test-Driven Interactive Code Generation (TiCoder)	Introduced user-in-the-loop test-driven workflows. Requires continuous human interaction, limiting scalability.
9	An LLM-Based Agent-Oriented Approach for Automated Code Design Issue Localization	Focused on design quality issues rather than hallucination detection or trust estimation.
10	Exploring the Effectiveness of LLM-based Test-Driven Interactive Code Generation	Evaluated interactive test generation through user studies, where trust is inferred indirectly. Our framework introduces an explicit numerical Trust Score for objective and automated trust estimation.

3 PROPOSED METHODOLOGY

3.1 System Architecture

A Trust Metric System is a proposed methodology that assesses the authenticity and trustworthiness of AI/LLM-created responses. As seen in Fig. 1, the Trust Metric System comprises of a Retrieval-Augmented Generation (RAG) component, followed by a validation pipeline consisting of multimodal inputs, and a hybrid learning process for trust score value creation.The process starts when a user submits a query or programming task. The query and/or programming task is sent to a Retrieval-Augmented Generation (RAG) component. The purpose of the RAG is to retrieve and deliver relevant trusted information from the internet's curated external knowledge sources: Documentation of technology, verified repository sites, or benchmark data sets. The information is used as the basis for generating the output, thereby proactively minimising the number of hallucinations that occur during the generation of the output.Once the information is retrieved, the LLM processes the information and generates two separate outputs: 1) The generated code; 2) The generated natural language summary/explanation. The generated output is sent through the multimodal validation framework independently, which will assess the different aspects of trustworthiness of the generated code and natural language summary/explanation. The process for verifying code validity has two stages. The first stage examines generated code for token level and execution level features similar to those found in benchmarking systems, COLLU-BENCH. In this phase, the system extracts Token-level log probabilities, Token types, Execution Feedback i.e., run-time errors or exceptions, which allows it to recognise logical inconsistencies found in the code as well as detect "dead code" and "hallucinations' in the code. The second stage examines the generated explanation for faithfulness. The verification approach employs an Entity Based Verification strategy based on Entity Transfer Functions (ETF's). In this case, the Entity objects are taken from both the code and its summary. The Stage compares the correctness, presence and alignment of Intent of the respective entities found in the code and summary to determine that there are no "extrinsic" or "intrinsic" hallucinations. The Next step for the AI to determine that APIs used for generating the AI's output generated code were valid and applied using CloudAPIBench principles in terms of valid APIs, parameters, and method calls. The end result is a fusion of the results obtained from the three validation pipelines, which is passed to a Hybrid CNN–MLP model to predict the Hallucination Probabilities for generating code and to produce a Trust Score that expresses the reliability of the AI's output or response.

3.2 Implementation Strategy

This study follows a systematic and incremental implementation strategy designed to ensure modularity, scalability, and interpretability of the proposed Trust Metric System. The overall workflow is divided into five well-defined phases, each addressing a specific component of the framework.

3.2.1 Phase 1: Collection of Aggregate Dataset. Each data instance in the dataset consists of multiple components that jointly represent an LLM-generated response and its validation context. Specifically, every data point includes:

- The user's request or programming task,
- Source code generated by a Large Language Model,
- A natural language summary or explanation of the generated code,
- Token-level characteristics and execution feedback obtained from static and dynamic code analysis,

- Entity-level correctness labels derived from summary validation,
- API correctness indicators reflecting the validity of API usage.

To ensure diversity, consistency, and reliable labeling, the dataset is constructed using publicly available benchmarks such as HumanEval and MBPP, along with custom-generated hallucination datasets.

3.2.2 Phase 2: Multi-Modal Feature Engineering. In this phase, three distinct feature vectors are constructed to capture different dimensions of trustworthiness:

- **Code Feature Vector**: Sequential token representations derived from token types and token-level log probabilities.
- **Execution Feature Vector**: Fixed-length numerical representations obtained from execution feedback such as runtime errors, exceptions, and output mismatches.
- **Summary Feature Vector**: Metrics measuring summary faithfulness, including the ratio of mapped to unmapped entities and the proportion of incorrect intent entities.
- **API Feature Vector**: API correctness represented as binary or categorical indicators.

These feature vectors jointly capture the syntactic, semantic, and functional correctness of the generated code.

3.2.3 Phase 3: Hybrid CNN–MLP Model Design. A hybrid neural architecture is employed to process heterogeneous feature types. The architecture consists of the following components:

- **CNN Branch**: A convolutional neural network processes sequential token-level features to identify local hallucination patterns such as abrupt probability drops and anomalous token sequences.
- **MLP Branch**: A multi-layer perceptron analyzes non-sequential numerical features derived from summary validation and API correctness.
- **Fusion Layer**: Outputs from the CNN and MLP branches are concatenated and passed to fully connected layers to learn cross-modal correlations.

The output layer predicts two normalized probabilities using sigmoid activation functions:

- Probability of code hallucination,
- Probability of summary hallucination.

3.2.4 Phase 4: Trust Score Formulation. The final Trust Score is computed as a weighted combination of validation outputs:

$$\text{Trust Score} = w_1(1 - P_{\text{code}}) + w_2(1 - P_{\text{summary}}) + w_3 S_{\text{API}},$$

where:

- P_{code} denotes the predicted probability of code hallucination,
- P_{summary} denotes the predicted probability of summary hallucination,
- S_{API} represents the normalized API correctness score,
- w_1, w_2, w_3 are tunable weights such that $w_1 + w_2 + w_3 = 1$.

This formulation converts complex multi-modal validation signals into a single, transparent, and user-interpretable trust metric.

3.2.5 Phase 5: Evaluation Methodology. The effectiveness of the proposed system is evaluated using both component-level and holistic metrics:

- **Hallucination Detection Metrics**: Precision, Recall, and F1-score are used to evaluate code and summary hallucination detection.
- **Trust Score Validation**: Spearman rank correlation is used to measure alignment between the generated Trust Score and expert human judgments.

Strong correlation with expert evaluations indicates that the proposed Trust Score reliably reflects developer intuition and serves as a practical measure of trustworthiness in AI-assisted software development tools.

4 RESULTS AND DISCUSSION

This section evaluates the effectiveness of the proposed Trust Metric System in detecting hallucinations and quantifying the trustworthiness of AI- and LLM-generated code and summaries. Experimental analysis is conducted on benchmark datasets using multiple validation pipelines to assess both component-level performance and the reliability of the aggregated Trust Score.

4.1 LLM Input–Output Analysis

The system accepts a programming-related query as input and produces two outputs:

- Source code generated by a Large Language Model,
- A natural language summary describing the generated code.

Initial observations indicate that LLM-generated outputs generally exhibit syntactic correctness and coherent formatting. However, hallucinations are often subtle and complex, making them difficult to detect using execution-only validation. Logical inconsistencies in API usage and discrepancies between code behavior and generated summaries were frequently observed. These findings highlight the necessity of a multi-modal evaluation approach to reliably assess trustworthiness.

4.2 Code Validation Results

The code validation pipeline identifies hallucination-prone patterns using two primary sources of evidence:

- Token-level probabilities and token sequence analysis,
- Execution feedback, including runtime errors and unexpected behaviors.

Experimental results demonstrate that:

- The CNN-based token analysis effectively identifies anomalous token patterns indicative of hallucinations.
- Execution feedback captures explicit runtime failures but fails to detect hallucinations that do not prevent successful execution.
- Token-level analysis significantly enhances hallucination detection beyond execution-only methods.
- Combining explicit (execution errors) and implicit (probability anomalies) signals increases overall detection coverage.

Compared to prior approaches relying solely on execution-based validation, the proposed method identifies a substantially higher number of hallucinated code samples.

4.3 Summary Validation Results

Summary validation employs an entity-based verification strategy inspired by Entity Transfer Functions (ETF) to assess faithfulness between generated code and its explanation. Experimental findings reveal:

- A high prevalence of extrinsic hallucinations, where entities referenced in summaries do not exist in the corresponding code.
- The presence of intrinsic hallucinations, where existing entities are mischaracterized or assigned incorrect intent.
- Entity-level metrics, such as mapped-to-unmapped entity ratios, effectively quantify summary faithfulness.

These results confirm that successful code execution does not guarantee summary correctness, underscoring the importance of independent summary validation.

4.4 API Usage Validation Results

API usage validation is performed using principles derived from CloudAPIBench. The results indicate that:

- LLMs perform reliably when generating high-frequency and commonly used APIs.
- Hallucinations frequently occur in low-frequency or domain-specific APIs, particularly in parameter selection and method invocation.
- Many API hallucinations do not trigger runtime errors, allowing incorrect usage to pass unnoticed during execution.

The API validation module successfully detects such issues and contributes critical signals to the overall Trust Score computation.

4.5 Hybrid Model Performance

The hybrid CNN–MLP architecture demonstrates strong performance by integrating multiple validation modalities:

- The CNN branch captures sequential token-level code patterns.
- The MLP branch effectively analyzes numerical features from summary validation and API correctness.
- Feature fusion enables detection of cross-modal hallucination patterns that are not identifiable through individual modalities.

The model achieves a balanced trade-off between precision and recall for both code and summary hallucination detection, ensuring stable and consistent performance across tasks.

4.6 Trust Score Evaluation

The Trust Score provides a unified and interpretable measure of output reliability by aggregating:

- Code generation correctness,
- Summary faithfulness,
- API usage validity.

Key observations include:

- Higher Trust Scores correlate with lower hallucination probabilities.
- Outputs with minor semantic or logical flaws receive lower Trust Scores despite successful execution.
- Outputs containing severe hallucinations consistently yield significantly lower Trust Scores.

Correlation analysis with expert judgments (using Spearman rank correlation) demonstrates strong alignment between Trust Scores and human intuition regarding code reliability.

4.7 Comparative Discussion

Compared to existing approaches, the proposed framework:

- Eliminates reliance on execution-only validation,
- Avoids dependence on continuous human feedback,
- Surpasses single-modality hallucination detection techniques.

By providing a fully automated, scalable, and interpretable trust evaluation mechanism, the Trust Metric System offers a practical alternative to interactive or human-in-the-loop methods.

4.8 Discussion Summary

Overall, experimental findings demonstrate that:

- Hallucinations in AI-generated code manifest across multiple dimensions and cannot be detected using a single validation strategy.
- Multi-modal evaluation significantly improves hallucination detection accuracy.
- The proposed Trust Score serves as an actionable and transparent indicator of trustworthiness.

The proposed system effectively addresses the limitations of existing hallucination detection methods and provides a robust solution for deploying LLMs in real-world, high-stakes software development environments.

5 CONCLUSION

In this study, the author has developed a comprehensive, explainable method of judging the trustworthiness of responses produced by AI and LLM systems, specifically in relation to the serious issue of "hallucinations" in both generated code and natural language summaries. By using a single, integrated system of operating, the Author has been able to combine Retrieval-Augmented Generation (RAG) with a multi-layered post-Generation Validation (PGV) for a thorough evaluation of an 'RAG'-based solution's reliability.

The results of these experiments show that the hallucinations produced by LLMs are multi-dimensional and that they cannot be detected accurately through either execution-based validation methods or single-modality checks. By using token-level (code), entity-level (summary), and API-level (validation) analysis through a hybrid powder CNN-MLP architecture, the Author's system is able to identify problematic and inconsistent elements in LLMs generated outputs. As a result, the Trust Score produced by this system is a single measure of trustworthiness that correlates well with expert judgments and improves the transparency, accountability, and user confidence in AI-generated outputs.

In conclusion, this research represents a viable and scalable means of providing a method for determining the level of trustworthiness of AI-assisted software development efforts. Future research could expand the scope of this work and result in the development of a complete framework for all programming languages, as well as include an analysis of security vulnerabilities and performance profiling, in addition to integrating the Trust Score.

6 REFERENCES

[1] F. Liu, Y. Pei, S. Alamir, and X. Liu. 2025. Exploring and evaluating hallucinations in LLM-powered code generation. *IEEE Transactions on Software Engineering*. To appear.

[2] K. Maharaj, R. Ramachandran, A. Arcuri, and S. McIntosh. 2023. ETF: An entity tracing framework for hallucination detection in code summaries. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '23)*. IEEE/ACM.

[3] T. N. Jiang, Q. Li, L. Tan, and Z. Zhang. 2024. COLLU-BENCH: A benchmark for predicting language model hallucinations in code. *arXiv preprint* arXiv:2408.08333.

[4] N. Jain, B. Ray, R. Kwiatkowski, V. Kumar, and M. Ramanathan. 2024. On mitigating code LLM hallucinations with API documentation. *arXiv preprint* arXiv:2405.00253.

[5] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, and S. K. Lahiri. 2024. Exploring the effectiveness of LLM-based test-driven interactive code generation. In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE '24)*. IEEE/ACM.

[6] Y. Tian, Z. Zhang, J. Chen, and Z. Zheng. 2025. CodeHalu: Investigating code hallucinations in LLMs via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '25)*. AAAI Press.

[7] Z. Zhang, Y. Wang, J. Chen, and Z. Zheng. 2025. LLM hallucinations in code generation: Mechanism, phenomena, and mitigation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '25)*. ACM.

[8] V. Agarwal, Y. Pei, S. Alamir, and X. Liu. 2025. CodeMirage: Hallucinations in code generated by large language models. *IEEE Transactions on Software Engineering*. To appear.

[9] F. Batole, D. O'Brien, R. Dyer, T. N. Nguyen, and H. Rajan. 2025. An LLM-based agent-oriented approach for automated code design issue localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '25)*. IEEE.

[10] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, M. Musuvathi, and S. K. Lahiri. 2024. LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Software* 41, 2 (2024), 45–53.