Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation

Thomas Jean-Michel Valentin[1], **Ardi Madadi**[2], **Gaetano Sapia**[2], **Marcel Böhme**[2]

[1] ENS Paris-Saclay, France
[2] Max Planck Institute for Security and Privacy, Germany
thomas.valentin@ens-paris-saclay.fr, {ardi.madadi, gaetano.sapia, marcel.boehme}@mpi-sp.org

Abstract

Generating code from a natural language programming task is one of the most successful applications of Large Language Models (LLMs). Yet, the generated program may be buggy. Without an oracle, such as an existing, correct implementation or a formal specification, can we somehow estimate how likely the generated program is correct?

In this paper, we propose a measure of incorrectness, called *incoherence*, that can be estimated efficiently in the absence of an oracle and allows us to establish a lower bound on the error, i.e., the probability that the LLM-generated program for that specification is incorrect. In our experiments, our incoherence-based methodology can automatically identify about two-thirds of incorrect programs without reports of false positives for the average task. In fact, *an oracle-based evaluation of LLMs can be reliably replaced by an incoherence-based evaluation*. In particular, we find a very strong agreement between the ranking of LLMs by the number of programs deemed correct via an oracle (pass@1) and the ranking of LLMs by the number of programs deemed correct via incoherence.

Data & analysis — https://github.com/mpi-softsec/difftrust
Extended version — https://arxiv.org/abs/2507.00057

1 Introduction

LLMs have demonstrated remarkable performance on code generation tasks. Yet, confabulation remains a key concern. Models often produce syntactically correct but functionally incorrect code, raising the critical question of when such outputs can be trusted. For instance, Fan et al. (2023) found that the vast majority of auto-generated programs for easy to medium LeetCode programming tasks are incorrect and explain that 57% of those do not even properly implement the task (“algorithmic misalignment”) while another 19% can only be fixed by changing multiple different code locations (multi-hunk). Pearce et al. (2025) analyzed code generated in scenarios relevant to high-risk cybersecurity weaknesses and found that 40% of the 1.7k LLM-generated programs actually contain security vulnerabilities.

While ground truth implementations or regression test suites provide a post-hoc evaluation of the generated code, they are often unavailable in real-world deployments, motivating the need for *correctness proxies*—that is, mechanisms that can flag potential failures without external supervision.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

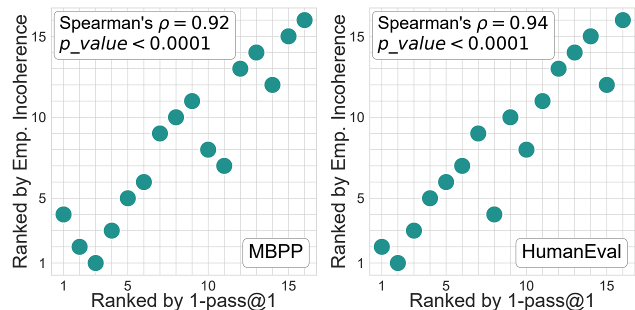

Figure 1: Rankings of 16 LLMs on the two most popular code generation benchmarks, MBPP and HumanEval. “Rank 1” indicates the highest probability of producing correct programs. *X-axis*: Ranking in terms of $[1 - \texttt{pass@1}]$ (i.e., the proportion of tasks with non-zero empirical error). *Y-axis*: Ranking in terms of the proportion of tasks with non-zero empirical incoherence. Note that incoherence can be estimated in the absence of a ground truth implementation.

> *Can we estimate how likely an LLM-generated program is correct in the absence of an oracle?*

Our work continues a recent stream of works addressing the confabulation problem using the *disagreement* between independently sampled responses to detect untruthful or erroneous outputs (Manakul, Liusie, and Gales 2023; Friel and Sanyal 2023; Li et al. 2024; Farquhar et al. 2024). A high disagreement indicates a high factual inconsistency.

However, existing measures of disagreement provide *no guarantees*; they are fundamentally heuristic in nature. Crucially, they can struggle to distinguish between confidently incorrect answers and correct answers generated under uncertainty—especially in complex structured domains like code, where semantics are hard to capture and where correctness is binary and unambiguous.

Our key insight is that—in the domain of code—the disagreement between independently sampled solutions for a task can be interpreted *semantically*: if two LLM-generated programs behave differently on the same input, at least one

must be incorrect. If the two programs behave identically across a representative input distribution, we gain empirical confidence in their correctness. This enables a shift from heuristic proxies to semantically grounded ones.

In this work, we formalize this intuition. We argue that incoherence, i.e., the behavioral divergence across samples, is a principled and theoretically justified proxy for model error. Concretely, given an LLM and a programming task d, we call the probability that any two programs generated to implement d are functionally different as the LLM's *incoherence* on d. If we are also given a ground truth implementation f_d^* for d, we call the probability that f_d^* and a program generated to implement d are functionally different as the LLM's *error* on d. We note that the widely-used `pass@1` score (Chen et al. 2021; Liu et al. 2023) on a benchmark set can be written in terms of the empirical error (i.e., the error's maximum likelihood estimate) on each task.

We develop a probabilistic framework that establishes a lower bound on the model's error in terms of incoherence. In contrast to prior work that relies on shallow patterns or internal model metrics, our approach directly leverages the one setting where semantic equivalence is exactly observable: executable code.

In experiments with 16 state-of-the-art LLMs and two popular code generation benchmarks, our incoherence measure, which requires *no* ground truth implementation, works incredibly well as a substitute for `pass@1`. Figure 1 shows rankings of those LLMs, both in terms of the proportion of tasks with non-zero empirical error (i.e., $[1 - \texttt{pass@1}]$) and the proportion of tasks with non-zero empirical incoherence. These rankings very strongly agree despite the absence of oracles for our incoherence measure ($\rho \geq 0.92$). We also find that a non-zero incoherence effectively detects about two-thirds of the non-zero errors in the absence of a ground truth implementation (69% and 66% detection rate on MBPP and HumanEval, resp.). No false positives. In cases where the incoherence is zero, the mean error is substantially lower than the average. If we increase the number of generated programs for a programming task 5-fold (from 10 to 50), the detection rate further increases by eight (8) percentage points—of course, at the cost of a 5-fold increase in monetary expenses for additional queries to the LLM.

Contributions:

1. We propose *incoherence*, a formal, unsupervised proxy for correctness that lower-bounds LLM error.
2. We develop a probabilistic framework linking incoherence to pass@1, with PAC-style efficiency guarantees.
3. Our study shows, incoherence detects errors reliably and yields rankings that agree with oracle-based evaluations.
4. We release all code, results, and analysis scripts.

2 Background

Code generation is a primary application of LLMs in software engineering (Hou et al. 2024). Tools like Copilot have seen widespread adoption, with over 40 million installations. Today, even general-purpose LLMs are competitive coders (Leaderboards 2025). In June 2025, nearly 25% of the 7.5 trillion tokens on (OpenRouter.ai 2025) were coding-related.

Jiang et al. (2024) highlight that *trustworthiness* is essential for LLM adoption in programming. Yet, LLMs are prone to generating incorrect code. How can correctness be validated without ground truth? Existing work addresses factuality in text by sampling multiple responses and measuring internal consistency—e.g., SelfCheckGPT (Manakul, Liusie, and Gales 2023), ChainPoll (Friel and Sanyal 2023), and semantic entropy approaches (Farquhar et al. 2024). These methods cluster outputs and estimate confidence based on entropy or token overlap. For code generation, variants include HonestCoder (Li et al. 2024), which uses syntax and data-flow modalities, and functional equivalence methods using symbolic execution (Sharma and David 2025). However, these approaches are fundamentally heuristic and lack formal guarantees. Their reliance on patterns or output similarity do not capture semantic correctness.

Mahmud et al. (2025) are interested in generating the most likely correct program for a task, and propose to select from a pool of candidates generated by multiple LLMs that program which best aligns with consensus, where consensus is defined both syntactically and semantically.

Our work addresses the *oracle problem* in software testing (Barr et al. 2015). While runtime crashes are detectable via sanitizers (Serebryany et al. 2012), functional correctness is domain-specific and often unobservable. We address this problem by viewing the generated program as a random variable and using incoherence as a proxy for correctness.

3 Defining Error and Incoherence

We consider the task of automatically generating a program from a natural language specification. Formally, given a textual description d of a programming task, a code generation system Coder—treated as a black-box stochastic process—samples a program $\pi \sim \mathsf{Coder}(d)$ intended to satisfy the task description d. Our objective is to assess the correctness of these programs $\mathsf{Coder}(d)$ without supervision, reference solutions, or access to model internals.

3.1 Notation

We write $\mathbb{P}(\cdot)$ for probability and $\mathbb{E}[\cdot]$ for expectation, leaving the underlying probability space implicit. For any expression *expr* involving random variables, $\mathbb{P}(\textit{expr})$ denotes the probability of the corresponding event. We use $\mathbb{I}(\textit{expr})$ for the indicator function, which is 1 when *expr* holds and 0 otherwise.

We denote by Descr the set of textual function descriptions and by Prog be the set of programs that define a function. Let $[\![\cdot]\!]$ denote the operational semantics such that for all $\pi \in \mathsf{Prog}$, $[\![\pi]\!]$ represents the function defined by π. We refer to $[\![\pi]\!]$ as the functional interpretation of π.

3.2 Error of a Code Generation System

We model a code generation system Coder as a function that maps each task $d \in \mathsf{Descr}$ to a corresponding (unknown) distribution over Prog. Formally, for all $d \in \mathsf{Descr}$:

$$\mathsf{Coder}(d) : \pi \in \mathsf{Prog} \mapsto p_\pi^d \in [0, 1] \qquad (1)$$

where p^d_π is the probability of obtaining π when querying Coder with task d. A program sampled from Coder for the task d is thus modelled by a random variable that follows the $\mathsf{Coder}(d)$ distribution:

$$\Pi^d \sim \mathsf{Coder}(d). \tag{2}$$

For every task description $d \in \mathsf{Descr}$, we assume there exist an input set Input_d, an output set Output_d and a correct (deterministic) ground truth implementation $\pi^*_d \in \mathsf{Prog}$ with its functional interpretation $f^*_d := [\![\pi^*_d]\!]$ such that $f^*_d : \mathsf{Input}_d \to \mathsf{Output}_d$.

The `pass@1` score (Chen et al. 2021) is a standard metric to evaluate the performance of Coder. For a finite set of tasks $S \subset \mathsf{Descr}$, `pass@1` is defined as the expected fraction of sampled programs that are functionally equivalent to the ground truth implementation:

$$\texttt{pass@1}(S) := \mathbb{E}\left[\frac{1}{|S|}\sum_{d\in S}\mathbb{I}\big([\![\Pi^d]\!] = f^*_d\big)\right]. \tag{3}$$

We define the **functional error** of Coder on task d as the complement of `pass@1` computed for a single task d, i.e., the probability that the generated program is not functionally equivalent to the ground truth:

$$\mathcal{E}(d) := \mathbb{P}\left([\![\Pi^d]\!] \neq f^*_d\right) = 1 - \texttt{pass@1}(\{d\}). \tag{4}$$

This definition captures the natural notion of error.

Moreover, we introduce a *probabilistic interpretation of correctness* with respect to (w.r.t.) a distribution of inputs. Rather than asking whether the generated function is correct for *all* inputs, which is undecidable due to Rice's theorem, we ask whether it is correct for a *typical input*, drawn from a distribution that represents expected usage. We use this probabilistic interpretation of correctness to introduce a pointwise notion of the error such that *a non-zero pointwise error implies a non-zero functional error* (cf. Eq. (4)).

We model an input generation system Gen as a function that maps each task $d \in \mathsf{Descr}$ to an (unknown) probability distribution over the corresponding input set Input_d. This distribution might represent how the program is executed under a typical workload. Formally, for all $d \in \mathsf{Descr}$:

$$\mathsf{Gen}(d) : x \in \mathsf{Input}_d \mapsto p^d_x \in [0,1] \tag{5}$$

where p^d_x intuitively models how likely is a function for d to be called on input $x \in \mathsf{Input}_d$.

We define the **pointwise error** of Coder w.r.t. Gen for any task $d \in \mathsf{Descr}$ as

$$\mathcal{E}_{\mathsf{Gen}}(d) := \mathbb{P}([\![\Pi^d]\!](X) \neq f^*_d(X)) \tag{6}$$

where $X \sim \mathsf{Gen}(d)$.

While the functional error can be computed only by verification of functional equivalence (an undecidable problem), the pointwise error can be estimated efficiently (c.f. Appendix C.1)—*in the presence of the oracle* f^*_d.

We note that a non-zero pointwise error implies a non-zero functional error, i.e.,

$$(\mathcal{E}_{\mathsf{Gen}}(d) > 0) \implies (\mathcal{E}(d) > 0). \tag{7}$$

Our pointwise error $\mathcal{E}_{\mathsf{Gen}}(d)$ models the practical reality that a program might be correct on almost all inputs that are empirically observed when the program is tested, deployed, or used in practice. By evaluating the probability of failure on a representative input distribution, the pointwise error provides a meaningful and practical estimate of the model's reliability in practical scenarios. The pointwise error also formalizes the experimental setup originally proposed and now widely used to estimate `pass@1` (i.e., the complement of the mean functional error on a fixed set of programming tasks) using a fixed set of random test cases (Chen et al. 2021; Liu et al. 2023).

4 Incoherence of a Code Generation System

Our core challenge is to estimate the pointwise error $\mathcal{E}_{\mathsf{Gen}}(d)$ *in the absence of the oracle* f^*_d, i.e., without supervision. We aim to achieve this using only observations from sampled implementations, without relying on any internal details of Coder. To this end, we specialize the disagreement-based hallucination detection approach (Manakul, Liusie, and Gales 2023) to the domain of code generation. The precise definition of the (probabilistic) correctness of a program w.r.t. an oracle (i.e., a ground truth implementation) provides us with the unique opportunity to formalize the approach and to introduce actual *probabilistic guarantees*.

We define the **pointwise incoherence** of Coder w.r.t. an input generation system Gen and a task d as the probability that two independently sampled programs produce different outputs on a generated input, i.e.,

$$\mathcal{I}_{\mathsf{Gen}}(d) := \mathbb{P}\left([\![\Pi^d_1]\!](X) \neq [\![\Pi^d_2]\!](X)\right) \tag{8}$$

where $\Pi^d_1, \Pi^d_2 \overset{iid}{\sim} \mathsf{Coder}(d)$ are two independently sampled programs and $X \sim \mathsf{Gen}(d)$ is an input sampled from Gen for task d. This quantity captures the model's internal uncertainty as revealed through behavioral divergence. Crucially, $\mathcal{I}_{\mathsf{Gen}}(d)$ is fully observable and efficient to estimate without an oracle (see Appendix C.2).

In the following, we show that this notion of pointwise incoherence provides a rigorous lower bound on the pointwise error and that it can be efficiently estimated. In Appendix D, we develop the notion of functional incoherence and establish a lower bound on the functional error in terms of the functional incoherence in parallel.

4.1 Lower Bound on Error in Terms of Incoherence

The pointwise incoherence provides a lower bound on the pointwise error. Intuitively, if two programs disagree on an input, at least one must be wrong; therefore, the probability of disagreement places a floor on the probability of failure.

Theorem 4.1 (Pointwise Incoherence Inequality)**.**

$$\forall \mathsf{Gen}, \forall d \in \mathsf{Descr}, \quad \mathcal{I}_{\mathsf{Gen}}(d) \leq 2 \times \mathcal{E}_{\mathsf{Gen}}(d).$$

Proof. See Appendix B. □

This result establishes $\mathcal{I}_{\mathsf{Gen}}(d)$ as a sound and theoretically grounded proxy for estimating model error on d. Unlike heuristic confidence or divergence metrics based on representation-level similarity, $\mathcal{I}_{\mathsf{Gen}}(d)$ directly, precisely, and formally captures observable functional disagreement.

Crucially, an error detection method based on our incoherence metric *never produces false positives*. The inequality guarantees that if the model has zero pointwise error on a task—i.e., $\mathcal{E}_{\mathsf{Gen}}(d) = 0$—then its pointwise incoherence must also be zero: $\mathcal{I}_{\mathsf{Gen}}(d) = 0$. This property distinguishes it from all previously proposed unsupervised proxies, which may still flag "uncertainty" even when outputs are correct.

4.2 Incoherence is Efficiently Estimated

A key advantage of incoherence as a surrogate for correctness is that it can be estimated efficiently and without access to ground-truth implementations. In this section, we formalize this claim by showing that both the pointwise incoherence and the decision problem of detecting non-zero incoherence admit simple, sample-efficient Monte Carlo estimators with standard PAC-style guarantees.

Theorem 4.2 (PAC Estimation)**.** *There exists a randomized algorithm that, given parameters $\delta > 0$, $\epsilon > 0$, code generator* Coder*, input generator* Gen*, and task $d \in D$, computes $\bar{\mathcal{I}}_{\mathsf{Gen}}(d)$ such that $\mathbb{P}(|\bar{\mathcal{I}}_{\mathsf{Gen}}(d) - \mathcal{I}_{\mathsf{Gen}}(d)| \leq \epsilon) \geq 1 - \delta$ using at most $\left\lceil \frac{\log(2/\delta)}{2\epsilon^2} \right\rceil$ samples.*

A similar theorem for the pointwise error, the randomized algorithms using Monte Carlo estimation, and the proofs for both theorems using a trivial application of Hoeffdings inequality are postponed to Appendix C.1& C.2.

If we are only interested in the decision problem using a boolean interpretation of correctness, a detection method offers a statistically sound and substantially more sample-efficient means to certify that Coder generates correct programs for a task d w.r.t. a well-specified usage distribution.

Theorem 4.3 (PAC Detection)**.** *There exists a randomized algorithm that, given parameters $\delta > 0$, $\epsilon > 0$, code generator* Coder*, input generator* Gen*, and task $d \in$* Descr*, returns* `true` *if a disagreement is observed and* `false` *otherwise, such that:*

- *If the algorithm returns* `true` *: $\mathcal{I}_{\mathsf{Gen}}(d) > 0$.*
- *If the algorithm returns* `false` *: $\mathcal{I}_{\mathsf{Gen}}(d) \leq \epsilon$ with probability at least $1 - \delta$,*

using at most $\left\lceil \frac{\log(\delta)}{\log(1-\epsilon)} \right\rceil$ samples.

A randomized algorithm based on Monte Carlo estimation and the proof is provided in Appendix C.3.

Implication for Error Detection. Although the algorithm described in Theorem 4.3 is designed to detect *non-zero incoherence*, we can use it to infer the presence of *non-zero error* due to the theoretical bound established in Theorem 4.1, which states:

$$\mathcal{I}_{\mathsf{Gen}}(d) \leq 2 \cdot \mathcal{E}_{\mathsf{Gen}}(d).$$

This implies that any task d for which $\mathcal{I}_{\mathsf{Gen}}(d) > 0$ must satisfy:

$$\mathcal{E}_{\mathsf{Gen}}(d) > 0.$$

Therefore, when the PAC detection algorithm returns `true` with high probability, we can conclude that the error rate is also bounded away from zero. This provides a conservative but sound certificate of model error without requiring access to a reference implementation.

5 Practical Considerations

5.1 Fixed Sampling Budget for $\mathsf{Coder}(d)$

In theory, pointwise incoherence can be estimated efficiently. The estimator defined by Equation (8) is easy to implement, parallelizable, and statistically robust. Voth incoherence estimation and detection admit PAC guarantees with low sample complexity (cf. Thm. 4.2, Thm. 4.3).

In practice, however, the primary bottleneck in large-scale evaluation is not sampling from the input distribution Gen, which is typically inexpensive, but generating programs from $\mathsf{Coder}(d)$, which typically requires querying an LLM. This cost can be substantial, especially when applied across a large set of tasks $d \in \mathsf{Descr}$.

To reduce this cost, we adopt a fixed sampling budget strategy: given a budget m, we draw m programs $\mathsf{Prog}_m = \langle \pi_1, \ldots, \pi_m \rangle$ once from $\mathsf{Coder}(d)$ and define an empirical code generator $\mathsf{Coder}_m(d)$ as the uniform distribution over these programs:

$$\mathsf{Coder}_m(d) := \text{Uniform}(\mathsf{Prog}_m).$$

This empirical generator approximates the original distribution $\mathsf{Coder}(d)$ while avoiding repeated expensive LLM queries at test time. As m increases, $\mathsf{Coder}_m(d)$ converges to $\mathsf{Coder}(d)$ in distribution, and the resulting estimates of incoherence become more faithful.

While this approximation introduces some additional variance, it is highly effective in practice. It amortizes LLM sampling costs across many evaluations, enabling scalable incoherence and error estimation. Experimentally, we find that a larger m consistently yields more reliable estimates.

5.2 Test Input Generation to Implement Gen

Automatic software test input generation is a well-studied problem in the software engineering community. Cast as a *constraint satisfaction problem*, we can use symbolic execution to generate inputs that exercise the different paths of a program (King 1976) or that reveal a difference between two program versions (Böhme, Oliveira, and Roychoudhury 2013). Cast as an *optimization problem*, we can use heuristic search to generate inputs that maximize code coverage (Ferguson and Korel 1996).

For our purposes, we propose to use *fuzzing*, an approach that mutates a set of user-provided or auto-generated seed inputs to generate new inputs. Today, fuzzing is the most successful and most widely-deployed automatic testing technique in practice (Böhme, Cadar, and Roychoudhury 2021). Like random test input generation, fuzzing is amenable to *statistical guarantees*, e.g., to quantify the probability of finding a bug with the next generated input in an ongoing testing campaign that has found no bugs (Böhme 2019, 2018; Böhme, Liyanage, and Wüstholz 2021; Lee and Böhme 2026, 2025, 2023).

In fact, fuzzing has recently been proposed specifically to improve the soundness of the evaluation of LLM-based code generators on the HumanEval and MBPP benchmarks (Liu et al. 2023), where `pass@1` (i.e., mean error across all benchmark tasks) was traditionally computed using five test inputs per task (Chen et al. 2021). The technique EvalPlus constructs the input distributions Gen in two stages:

1. **Seed Corpus Generation**: An LLM is prompted with the specification (or the ground truth implementation) to produce a set of canonical input examples.
2. **Type-Aware Mutation**: These examples are mutated using transformations that preserve the input types but introduce variation (e.g., altering values, shuffling list contents, varying string formats).

We observe that Stage 1 might introduce a bias where an LLM's generated code might appear to perform better on inputs generated by the same LLM, compared to inputs generated by another LLM. Hence, in our experiments, to provide a fair evaluation of all considered LLMs, we mitigate that potential bias by using the benchmark-provided test inputs as seed inputs. In practice, in the absence of existing test inputs, we suggest using the original method or discovering the seed corpus using greybox fuzzings (Zalewski 2014).

6 Experimental Setup

6.1 Research Questions

Our study aims to answer the following research questions.

- **RQ.1 (Effectiveness)**. How effectively can errors be detected using incoherence alone without an oracle? What is the average error when incoherence is zero? How strong is the relationship between incoherence and error?
- **RQ.2 (Agreement)**. Does the result of an incoherence-based evaluation agree with the result of an error-based evaluation of LLMs?
- **RQ.3 (Ablation)**. How do incoherence and error vary as a function of a) the number of synthesized programs, b) the number of generated inputs, or c) the temperature?

6.2 Models and Datasets

Claude 4 Opus (2025/05/14)	Claude 4 Sonnet (2025/05/14)
DeepSeek-Coder R1	DeepSeek-V3 (0324)
Gemini 2.0 Flash Lite	Gemini 2.5 Pro (preview 05/06)
Gemini 2.5 Flash (preview 05/20)	GPT-3.5 Turbo
GPT-4	GPT-4 Turbo
GPT-4o	GPT-o4 Mini
LLaMA 3.1 8B Instruct	LLaMA 3.3 70B Instruct
LLaMA 4 Maverick 17B	Ministral 8B

Table 1: Large Language Models used in our experiments.

Models. Table 1 shows the large language models (LLMs) used in our experiments. At the time of writing, these 16 LLMs represent the most successful LLMs for code generation according to several popular leaderboards (Leaderboards 2025). They also represent the current portfolio of the most popular LLM vendors: Anthropic, DeepSeek, Google, Meta, Mistral, and OpenAI. By default, we chose a *temperature of 0.6*, a value commonly used in prior work on code generation (Li et al. 2024; DeepSeek-AI et al. 2025). We vary the temperature parameter in the ablation study (RQ3).

Datasets. We evaluate our measures of incoherence and error using the 16 LLMs on two (2) popular code generation benchmarks: HumanEval (Ji et al. 2025) and MBPP (Mostly Basic Python Problems) (Hu et al. 2025). *HumanEval* is a human-written benchmark published by OpenAI in 2021, consisting of 164 programming tasks. *MBPP* is a crowd-sourced benchmark published by Google in 2022. We used the author-sanitized version of MBPP containing 426 hand-verified programming tasks. For every task, they offer

- a natural language description of the task d,
- a ground-truth Python implementation f_d^*, and
- an average of 7.7 (and 3) Python test inputs for HumanEval (and MBPP, respectively).

6.3 Variables and Measures

Given a code generator Coder and input generator Gen, programming task d, a query budget m and a testing budget n, the **empirical error** $\hat{\mathcal{E}}(d,m,n)$ on d as estimator of the pointwise error is computed as

$$\hat{\mathcal{E}}(d,m,n) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}\big([\![\pi_{y_i}^d]\!](x_i^d) \neq f_d^*(x_i^d)\big) \tag{9}$$

and the **empirical incoherence** $\hat{\mathcal{I}}(d,m,n)$ on d as estimator of the pointwise incoherence is computed as

$$\hat{\mathcal{I}}(d,m,n) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}\big([\![\pi_{y_i}^d]\!](x_i^d) \neq [\![\pi_{y_i'}^d]\!](x_i^d)\big) \tag{10}$$

where $\pi_1^d,...,\pi_m^d$ are sampled from $\mathsf{Coder}(d)$, $x_1^d,...,x_n^d$ are sampled from $\mathsf{Gen}(d)$ and $y_1,...,y_n,y_1',...,y_n'$ are sampled from Uniform$(\{1,...,m\})$.

Given the set of programming tasks S, we can now write the **empirical `pass@1`** score in terms of the empirical error (cf. Eq. (4)):

$$1 - \frac{1}{|S|}\sum_{d\in S}\mathbb{I}\big(0 \neq \hat{\mathcal{E}}(d,1,n)\big) \tag{11}$$

The **mean empirical error** $\bar{\mathcal{E}}(S,m,n)$ is $\sum_{d\in S}\frac{\hat{\mathcal{E}}(d,m,n)}{|S|}$ while the **mean empirical incoherence** $\bar{\mathcal{I}}(S,m,n)$ is computed as $\sum_{d\in S}\frac{\hat{\mathcal{I}}(d,m,n)}{|S|}$.

The **detection rate** is the proportion of tasks with non-zero emp. error that have a non-zero empirical incoherence, i.e., $\frac{1}{|S_x|}\sum_{d\in S_x}\mathbb{I}\big(0 \neq \hat{\mathcal{I}}(d,m,n)\big)$ where $S_x = \{d \mid d \in S \wedge \hat{\mathcal{E}}(d,m,n) \neq 0\}$.

The **undetected mean empirical error** is the mean empirical error of tasks with zero empirical incoherence, i.e., $\frac{1}{|S_u|}\sum_{d\in S_u}\hat{\mathcal{E}}(d,m,n)$ where $S_u = \{d \mid d \in S \wedge \hat{\mathcal{I}}(d,m,n) = 0\}$.

We measure the **strength of the relationship** between two random variables, i.e., empirical incoherence and error, using Spearman's rank correlation coefficient ρ. We measure the **agreement on ranking** when sorting the performance of LLMs measured by the proportion of programming tasks (a) with zero mean empirical error versus (b) with zero mean empirical incoherence, also using Spearman's rank correlation coefficient.

	LLM	Mean Error	Mean Incoherence	Spearman Correlation	Detection Rate	Undetected Mean Error
MBPP	Code	0.2960	0.0995	0.5276	0.6866	0.2071
	General	0.2773	0.1123	0.6105	0.7243	0.1638
	Small	0.3741	0.1641	0.5892	0.7037	0.2107
	Mean	0.3009	0.1203	0.5621	0.6857	0.1866
Human Eval	Code	0.0763	0.0295	0.7171	0.7188	0.0417
	General	0.0927	0.0483	0.7181	0.7042	0.0460
	Small	0.1585	0.0911	0.7282	0.7381	0.0737
	Mean	0.1050	0.0560	0.6861	0.6616	0.0471

Table 2: Performance of 3 LLMs on 2 benchmarks. The **mean** is reported across all 16 LLMs; `Code` = Gemini 2.5 Pro, `General` = Gpt–4o, and `Small` = Ministral 8b.

6.4 Implementation

For each task, we generate m candidate functions per coder (default $m = 10$), using vendor APIs. Inputs are generated via mutation-based fuzzing ($n = 1000$ by default). We estimate both error (w.r.t. ground truth f_d^*) and incoherence (between candidates), with all executions sandboxed (60s timeout). Experiments ran on AMD EPYC 7713P CPU (128 threads), 251 GB RAM. More details in Appendix A.

7 Empirical Results

RQ-1. Effectiveness

Table 2 shows the results for across all 16 LLMs (mean) and for three representative models (code, general, and small) for both code generation benchmarks. Table 4 (appendix) shows the results for all 16 LLMs. The measures in the header row are discussed in Section 6.3. Figure 2 shows a scatter plot illustrating the relationship between error and incoherence.

Results. A non-zero incoherence *effectively* detects a non-zero error without access to a ground truth implementation. The mean detection rate across all 16 LLMs for MBPP and HumanEval are 69% and 66%, respectively. There does not seem to be a substantial difference in detection rate between the code-generation specific LLM (Gemini 2.5 Pro) and the general-purpose or the small LLM (GPT-4o, Mistral 8b). In cases where the incoherence is zero, the mean error is also substantially lower. Concretely, the mean error reduces from 30% to 19% for MBPP and from 11% to 5% for HumanEval ("Undetected Mean Error"). For the small LLM, the mean error is generally higher than for the other LLMs, but the percentage decrease when incoherence is zero is similar.

Figure 2 best illustrates the relationship between error and incoherence. We can clearly see the consequence of the inequality in Theorem 4.1. Error is usually greater than incoherence for a programming task. There are some tasks (on the left of each plot) where incoherence is zero but the error is non-zero. For the average LLM, we find a *moderate correlation* between error and incoherence (0.56 for MBPP; 0.69 for HumanEval). In Table 2, for the three LLMs evaluated on HumanEval, we even find a *strong correlation* (gt. 0.7).

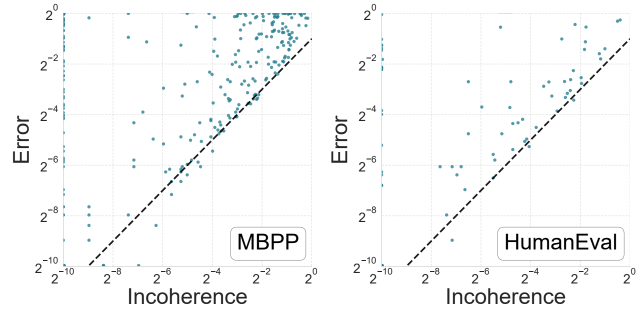

Figure 2: Relationship between error and incoherence for GPT-4o on MBPP and HumanEval benchmarks. The dashed line demonstrates the inequality in Theorem 4.1.

> **RQ-1**. *A non-zero incoherence effectively detects about two-thirds of the non-zero errors in the absence of a ground truth implementation. In cases where the incoherence is zero, the mean error is substantially lower than the average. The plot of error and incoherence provides empirical confirmation for our inequality.*

RQ-2. Agreement on LLM Ranking

If the incoherence- and error-based rankings of LLMs agree, we can reliably substitute one measure for the other. We could remove the requirement to provide painstakingly manually-written ground-truth implementations for every programming task when constructing new code generation benchmarks. We could *mitigate critical threats to validity* in benchmarking of new LLM-based code generation systems, such as overfitting or data leakage.

Figure 1 (on the title page) shows a scatter plot of the ranking of all 16 LLMs in terms of the number of projects with zero error (i.e., `pass@1`; cf. Eqn. (3)) versus the ranking of the same LLMs in terms of the number of projects with our non-zero oracle-less incoherence measure.

Results. We observe a *very strong agreement* on the rankings. Concretely, $\rho = 0.92$ and $\rho = 0.94$ for MBPP and HumanEval, respectively, at a significance level $p < 0.0001$. The rankings are close to the diagonal.

> **RQ-2**. *An oracle-based evaluation can be reliably substituted by an incoherence-based evaluation. Specifically, there is a* very strong agreement *between the rankings of LLMs in terms of the proportion of programming tasks that are considered correct (a) via the lack of a pointwise difference with a ground truth implementation (as in,* `pass@1`*) versus (b) via the lack of a pointwise difference between two randomly generated solutions.*

RQ-3. Ablation

Table 3 shows the results of our ablation study as we vary the LLM the number m of generated programs, the number n of generated test inputs, or the LLM's temperature t. A higher temperature increases the likelihood that the LLM samples a lower-probability token during next-token prediction. We

	MBPP		*HumanEval*	
	Expenses (in USD)	**Detection Rate**	**Expenses (in USD)**	**Detection Rate**
$m = 1$	0.8730	0.0000	0.4436	0.0000
$m = 2$	1.7557	0.3974	0.8904	0.3333
$m = 5$	4.3992	0.6357	2.2189	0.5846
$m = 10$	8.8138	0.7243	4.4530	0.7042
$m = 25$	22.0332	0.7742	11.1055	0.8267
$m = 50$	43.9539	0.8105	22.2106	0.8182

(a) Detection rate and LLM costs as the query budget, i.e., the number m of programs generated by $\mathsf{Coder}(d)$ *increases.*

$t = 0.2$	8.8138	0.5422	4.4530	0.5556
$t = 0.6$	8.8174	0.7148	4.4840	0.7286
$t = 1$	9.0657	0.8050	4.5254	0.7600

(b) Detection rate and LLM costs as temperature t of $\mathsf{Coder}(d)$ *increases.*

$n = 100$	8.8174	0.6811	4.4840	0.6615
$n = 1000$	8.8174	0.7148	4.4840	0.7286
$n = 2000$	8.8174	0.7200	4.4840	0.7083
$n = 5000$	8.8174	0.7355	4.4840	0.7222
$n = 10000$	8.8174	0.7445	4.4840	0.7222

(c) Detection rate and LLM costs as the testing budget, i.e., the number n of test inputs generated by $\mathsf{Gen}(d)$ *increases.*

Table 3: Results of our ablation study. We vary one value while keeping all others constant. Coder is GPT-4o. Default number of programs per task: $m = 10$. Default number of test inputs per task: $n = 1000$. Default temperature: $t = 0.6$.

vary one parameter and keep all others constant ($m = 10$, $n = 1000$, $t = 0.6$, GPT4-o; cf. §6).

Query budget m. The detection rate increases with m. For instance, when changing m from 10 to 50, the detection rate increases by 14–19% from 0.7 to about 0.83 for HumanEval and from 0.72 to 0.82 for MBPP. The advantage comes at a substantial monetary cost. When changing m from 10 to 50, our expenses increased by more than fivefold, e.g., from \$9 to \$44 for MBPP. For HumanEval, we actually observe a slightly higher detection rate (0.83) at $m = 25$, which we explain by the randomness of the sampling and test generation process. It might also indicate that the detection rate starts to saturate for larger values of m (which we determined as uneconomical for us to test). Another interesting observation is that just sampling a second program ($m = 2$) already gives us a 0.33 to 0.4 detection rate.

Temperature t. Detection rate increases with t. For instance, when changing t from 0.2 to 1.0, we see detection rate increase by 36–50% from 0.54 to 0.81 for MBPP and from 0.56 to 0.76 for HumanEval. A high temperature induces a high output diversity, which seems to increase the LLM's incoherence, which serves us well in error detection.

Test inputs n. Detection rate increases with n. However, compared to the other hyperparameters, a substantial increase in the number of generated test inputs induces only a relatively small increase in detection rate.

> **RQ-3**. *Increasing* Coder*'s query budget m,* Gen*'s testing budget n, or the temperature t also increases the detection rate. However, an x-fold increase in query budget comes at a greater-than x-fold increase in expenses.*

8 Threats to Validity

As with any empirical study, there are threats to the validity of our results and conclusions. The first threat is to the *external validity*, i.e., the extent to which our findings can be generalized. As the subjects of our study, we selected LLMs from all major LLM vendors that were top-performing according to code generation leaderboards. They represent the current state-of-the-art. As the objects of our study, we selected the two most widely used code generation benchmarks, MBPP and HumanEval, to facilitate comparison with results in related research. However, the findings may not generalize to more complex programming tasks or programming languages other than Python, and we call on the community to replicate our experiments for their use cases. Beyond the empirical results we also formally prove certain properties of incoherence and its estimation in the general.

The second threat is to the internal validity, i.e., the extent to which the presented evidence supports our claims about cause and effect within the context of our study. In the benchmarks, the task description may be ambiguous or the ground-truth implementation incorrect (Siddiq et al. 2024). We use popular well-scrutinized benchmarks. From MBPP, we chose the best-quality, hand-curated set of tasks. DiffTrustmay contain bugs itself, but we release all our scripts and data for the community to scrutinize.

9 Perspective

We believe that our incoherence-based perspective gives rise to a proliferation of new techniques built for *trustworthy code generation with probabilistic guarantees*.

In this paper, we discuss the formal estimation of the correctness of an LLM-generated program when there is *no automated mechanism* to decide whether a program is correct or not (e.g., a formal specification or a ground-truth implementation). We model the generated program as a random variable drawn from an unknown distribution induced by the coder (e.g., the LLM). This opens the door for a probabilistic notion of correctness. Our measure, incoherence, formalizes the observation that, if two random programs for the same task disagree on the output for an input, at least one must be incorrect. We formally demonstrate how the coder's error on a task has a lower bound in terms of the coder's incoherence on that task and empirically observe that a non-zero incoherence detects more than two-thirds of the incorrect programs. Since incoherence does not depend on model internals, it can be applied broadly across LLMs of any kind and even to probabilistic systems like LLM-agents.

An incoherence-based evaluation of the code generation capabilities of multiple LLMs also addresses several open challenges of the traditional ground-truth-based `pass@1` evaluation. The existing process of curating large coding benchmarks with correct human-generated ground-truth implementations is labour-intensive, error-prone, and subject to future data leakage issues (Ramos et al. 2025). Incoherence paves the way for evaluations on a substantially larger scale, basically on a stream of programming tasks.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and for helping us improve this paper. This research is partially funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is supported by ERC grant (Project AT_SCALE, 101179366). This research is also partially funded by the ENS Paris-Saclay who graciously supported the internship of the first author at the MPI for Security and Privacy. This research was conducted during an undergraduate internship of the first author at the MPI for Security and Privacy.

References

Barr, E. T.; Harman, M.; McMinn, P.; Shahbaz, M.; and Yoo, S. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5): 507–525.

Böhme, M. 2018. STADS: Software Testing as Species Discovery. *ACM Trans. Softw. Eng. Methodol.*, 27(2).

Böhme, M. 2019. Assurance in software testing: a roadmap. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '19, 5–8. IEEE Press.

Böhme, M.; Cadar, C.; and Roychoudhury, A. 2021. Fuzzing: Challenges and Reflections. *IEEE Software*, 38(3): 79–86.

Böhme, M.; Liyanage, D.; and Wüstholz, V. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 15th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, 494–504.

Böhme, M.; Oliveira, B. C. D. S.; and Roychoudhury, A. 2013. Partition-based regression verification. In *2013 35th International Conference on Software Engineering (ICSE)*, 302–311.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374.

DeepSeek-AI; Guo, D.; Yang, D.; Zhang, H.; Song, J.; Zhang, R.; Xu, R.; Zhu, Q.; Ma, S.; Wang, P.; Bi, X.; Zhang, X.; Yu, X.; Wu, Y.; Wu, Z. F.; Gou, Z.; Shao, Z.; Li, Z.; Gao, Z.; Liu, A.; Xue, B.; Wang, B.; Wu, B.; Feng, B.; Lu, C.; Zhao, C.; Deng, C.; Zhang, C.; Ruan, C.; Dai, D.; Chen, D.; Ji, D.; Li, E.; Lin, F.; Dai, F.; Luo, F.; Hao, G.; Chen, G.; Li, G.; Zhang, H.; Bao, H.; Xu, H.; Wang, H.; Ding, H.; Xin, H.; Gao, H.; Qu, H.; Li, H.; Guo, J.; Li, J.; Wang, J.; Chen, J.; Yuan, J.; Qiu, J.; Li, J.; Cai, J. L.; Ni, J.; Liang, J.; Chen, J.; Dong, K.; Hu, K.; Gao, K.; Guan, K.; Huang, K.; Yu, K.; Wang, L.; Zhang, L.; Zhao, L.; Wang, L.; Zhang, L.; Xu, L.; Xia, L.; Zhang, M.; Zhang, M.; Tang, M.; Li, M.; Wang, M.; Li, M.; Tian, N.; Huang, P.; Zhang, P.; Wang, Q.; Chen, Q.; Du, Q.; Ge, R.; Zhang, R.; Pan, R.; Wang, R.; Chen, R. J.; Jin, R. L.; Chen, R.; Lu, S.; Zhou, S.; Chen, S.; Ye, S.; Wang, S.; Yu, S.; Zhou, S.; Pan, S.; Li, S. S.; Zhou, S.; Wu, S.; Ye, S.; Yun, T.; Pei, T.; Sun, T.; Wang, T.; Zeng, W.; Zhao, W.; Liu, W.; Liang, W.; Gao, W.; Yu, W.; Zhang, W.; Xiao, W. L.; An, W.; Liu, X.; Wang, X.; Chen, X.; Nie, X.; Cheng, X.; Liu, X.; Xie, X.; Liu, X.; Yang, X.; Li, X.; Su, X.; Lin, X.; Li, X. Q.; Jin, X.; Shen, X.; Chen, X.; Sun, X.; Wang, X.; Song, X.; Zhou, X.; Wang, X.; Shan, X.; Li, Y. K.; Wang, Y. Q.; Wei, Y. X.; Zhang, Y.; Xu, Y.; Li, Y.; Zhao, Y.; Sun, Y.; Wang, Y.; Yu, Y.; Zhang, Y.; Shi, Y.; Xiong, Y.; He, Y.; Piao, Y.; Wang, Y.; Tan, Y.; Ma, Y.; Liu, Y.; Guo, Y.; Ou, Y.; Wang, Y.; Gong, Y.; Zou, Y.; He, Y.; Xiong, Y.; Luo, Y.; You, Y.; Liu, Y.; Zhou, Y.; Zhu, Y. X.; Xu, Y.; Huang, Y.; Li, Y.; Zheng, Y.; Zhu, Y.; Ma, Y.; Tang, Y.; Zha, Y.; Yan, Y.; Ren, Z. Z.; Ren, Z.; Sha, Z.; Fu, Z.; Xu, Z.; Xie, Z.; Zhang, Z.; Hao, Z.; Ma, Z.; Yan, Z.; Wu, Z.; Gu, Z.; Zhu, Z.; Liu, Z.; Li, Z.; Xie, Z.; Song, Z.; Pan, Z.; Huang, Z.; Xu, Z.; Zhang, Z.; and Zhang, Z. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

Fan, Z.; Gao, X.; Mirchev, M.; Roychoudhury, A.; and Tan, S. H. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, 1469–1481. IEEE Press. ISBN 9781665457019.

Farquhar, S.; Kossen, J.; Kuhn, L.; and Gal, Y. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630: 625–627.

Ferguson, R.; and Korel, B. 1996. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1): 63–86.

Friel, R.; and Sanyal, A. 2023. ChainPoll: A High Efficacy Method for LLM Hallucination Detection. *arXiv preprint arXiv:2310.18344*.

Hou, X.; Zhao, Y.; Liu, Y.; Yang, Z.; Wang, K.; Li, L.; Luo, X.; Lo, D.; Grundy, J.; and Wang, H. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.*, 33(8).

Hu, Y.; Zhou, Q.; Chen, Q.; Li, X.; Liu, L.; Zhang, D.; Kachroo, A.; Oz, T.; and Tripp, O. 2025. QualityFlow: An Agentic Workflow for Program Synthesis Controlled by LLM Quality Checks. *arXiv preprint arXiv:2501.17167*.

Ji, S.; Song, Z.; Zhong, F.; Jia, J.; Wu, Z.; Cao, Z.; and Xu, T. 2025. MyGO Multiplex CoT: A Method for Self-Reflection in Large Language Models via Double Chain of Thought Thinking. *arXiv preprint arXiv:2501.13117*.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024.

A Survey on Large Language Models for Code Generation. arXiv:2406.00515.

King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394.

Leaderboards, A. 2025. Various Leaderboards for LLM-based Code Generation. https://livecodebench.github.io/leaderboard.html https://evalplus.github.io/leaderboard.html https://bigcode-bench.github.io/.

Lee, S.; and Böhme, M. 2023. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, 12.

Lee, S.; and Böhme, M. 2025. How Much is Unseen Depends Chiefly on Information About the Seen. In *Proceedings of the 13th International Conference on Learning Representations*, ICLR'25.

Lee, S.; and Böhme, M. 2026. Dependency-aware Residual Risk Analysis. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering*, ICSE'26.

Li, J.; Zhu, Y.; Li, Y.; Li, G.; and Jin, Z. 2024. Showing LLM-Generated Code Selectively Based on Confidence of LLMs. *arXiv preprint arXiv:2410.03234*.

Liu, J.; Xia, C. S.; Wang, Y.; and ZHANG, L. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In Oh, A.; Naumann, T.; Globerson, A.; Saenko, K.; Hardt, M.; and Levine, S., eds., *Advances in Neural Information Processing Systems*, volume 36, 21558–21572. Curran Associates, Inc.

Mahmud, T.; Duan, B.; Pasareanu, C.; and Yang, G. 2025. Enhancing LLM Code Generation with Ensembles: A Similarity-Based Selection Approach. arXiv:2503.15838.

Manakul, P.; Liusie, A.; and Gales, M. J. F. 2023. SelfCheckGPT: Zero-Resource Black-Box Hallucination Detection for Generative Large Language Models. *arXiv preprint arXiv:2303.08896*.

OpenRouter.ai. 2025. Top LLMs in Programming. https://openrouter.ai/rankings/programming?view=month.

Pearce, H.; Ahmad, B.; Tan, B.; Dolan-Gavitt, B.; and Karri, R. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM*, 68(2): 96–105.

Ramos, D.; Mamede, C.; Jain, K.; Canelas, P.; Gamboa, C.; and Goues, C. L. 2025. Are Large Language Models Memorizing Bug Benchmarks? arXiv:2411.13323.

Serebryany, K.; Bruening, D.; Potapenko, A.; and Vyukov, D. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 28. USA: USENIX Association.

Sharma, A.; and David, C. 2025. Assessing Correctness in LLM-Based Code Generation via Uncertainty Estimation. arXiv:2502.11620.

Siddiq, M. L.; Dristi, S.; Saha, J.; and Santos, J. C. S. 2024. The Fault in our Stars: Quality Assessment of Code Generation Benchmarks. arXiv:2404.10155.

Zalewski, M. 2014. Pulling JPEGs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html.