How Robustly do LLMs Understand Execution Semantics?

Anonymous Author(s)

Abstract

LLMs demonstrate remarkable reasoning capabilities, yet whether they utilize internal world models or rely on sophisticated pattern matching remains open. We study LLMs through the lens of *robustness of their code understanding* using a standard program-output prediction task. Our results reveal a stark divergence in model behavior: while open-source reasoning models (DeepSeek-R1 family) maintain stable, albeit somewhat lower accuracies (38% to 67%) under code transformations & input perturbations, the frontier model GPT-5.2 exhibits significant brittleness. Despite achieving a near-perfect score of 99% on the original, unperturbed CruxEval benchmark, perturbed inputs trigger accuracy declines between 20% and 24%. In addition, we find that many models perform much worse at predicting behavior on perturbed inputs that raise exceptions, *and* that prediction performance depends on the kind of exception. We study remedies to address this deficiency in exception prediction, and evaluate the effect of these remedies on the ability to predict non-exception behaviors. Our findings both point to limitations in the way all models understand code, and establish the value of using perturbation to evaluate code models.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

code understanding, robustness, metamorphic testing

ACM Reference Format:

Anonymous Author(s). 2018. How Robustly do LLMs Understand Execution Semantics?. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Current LLMs show impressive proficiency in Software Engineering tasks, such as code generation [24], code summarization [38], and code repair [49]. This raises the question: to what extent does language model performance arise from actually *understanding* the semantics of code? Studies show that developers spend the majority of their time understanding existing code [25, 44]; the capacity for *understanding* code is thus vital to human coding work. For example, maintenance work can require understanding what a buggy program actually does, in response to an failure-triggering input. Thus, it is natural to ask if LLMs are similarly able to understand code.

Language model "understanding" is a topic well-explored for natural language, using benchmarks such as MMLU [18] and SuperGLUE [39]. Recently, several benchmarks have tackled this issue for code. CodeMMLU [30] comprises multiple-choice tests that evaluate several types of code understanding (syntax and semantics). CruxEval [15] is more demanding, requiring LLMs to understand programs well-enough to *precisely predict exact outputs* for given test inputs (not just select one answer from presented choices, as with CodeMMLU).

To effectively maintain a program, one must understand it sufficiently well to *robustly* and consistently predict its behavior, regardless of how exactly it is coded, which inputs it is given, or the intricacy of their execution paths. Predicting outputs given inputs is a vital capability for software maintenance tasks: developers need to do this when diagnosing failures, when refactoring the code or adding enhancements. LLMs perform surprisingly well on this task, with frontier models predicting outputs for inputs for published benchmarks [15] with over 99% accuracy, but how **robust** is this performance? In other words, how much can a developer rely on LLM predictions while debugging or maintaining code?

Prior work suggests that language model understanding of code is *not* robust with respect to program structure for highly demanding tasks like code equivalence and static analysis [20, 42]. In this paper, we aim to refine the benchmarks for evaluating LLM code understanding by treating output prediction as a proxy for code understanding. We define robustness as the ability of a model to consistently correctly predict a program's output on a given input despite variations that do not alter the program's semantics. To measure it, our study contributes an evaluation of LLM robustness across three critical dimensions:

First, we evaluate the robustness of LLM ability to predict program outputs, *when subject to input perturbation*. We apply type-aware mutation to generate a dense neighborhood of local inputs for each program; we find that LLMs frequently do not make consistent (all right or all wrong) output predictions, even within these constrained input spaces, indicating a weak understanding of the code. Some inputs cause a program to fail; the LLM should be able to correctly and consistently predict behavior for such failing runs as well.

> **RQ 1:** How robust is LLM code understanding to input perturbation? How well do LLMs predict runtime exceptions for perturbed inputs that cause exceptions?

Finding Summary: We find evidence of lack of robustness, both (unexpectedly) in some large, frontier models, *and* also with exception-raising inputs. We explore these issues in detail.

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

Next, we consider robustness of LLMs on the output prediction task, *when subject to program transformation*: we use meaning-preserving transformations (MPTs) to evaluate LLM performance on output prediction by comparing a model's performance on syntactically different, but semantically equivalent, variants of an initial program.

> **RQ 2:** How robust is LLM code understanding to program perturbation via meaning-preserving code transformations?

Finding Summary: With a few unexpected exceptions, notably in frontier models, we find that models mostly respond robustly to the meaning-preserving transformations in our study.

Finally, we study the robustness of LLMs, on the same output prediction task, *when execution traces have varying numbers of decisions*: we use dynamic analysis (tracing) to evaluate LLM performance on output prediction as a function of the number of decisions encountered during execution.

> **RQ 3:** How robust is LLM code understanding to control flow decisions? Are execution paths with more decisions, more difficult to reason about?

Finding Summary: In general, our data broadly suggests that LLMs are less accurate at predicting outputs as more decisions are encountered along an execution trace. When the decisions are primarily loop control conditions, as opposed to if-else chains, this finding points to disfluencies in LLM's understanding of iteration.

2 Background

To evaluate the robustness of LLMs on code execution tasks, we distinguish between standard performance and structural consistency. Given a set of programs $\mathcal{P}$, where each program $p \in \mathcal{P}$ is associated with a canonical input $x_{p,0}$ (from the original CruxEval benchmark) and a set of n synthetic variants $\{x_{p,1}, \ldots, x_{p,n}\}$, we define the following metrics:

2.1 Performance Metrics

- **Accuracy** C_o: The model's accuracy on the canonical CruxEval input set. It serves as the baseline for comparison with prior work, and is equivalent to Pass@1.
- **Accuracy** $C_{\tilde{o}}$: The model's average accuracy across all perturbed input variants CruxEval$_o$, excluding the original.
- **Accuracy** $C_{o \cup \tilde{o}}$: The mean accuracy across the union of original and perturbed inputs.

2.2 Robustness and Generalization

- **Program-level Strict Robustness (PSR)**: We introduce this new robustness measure. It is the fraction of programs on which a model predicts the correct output for *all* n inputs; PSR measures how likely a model is to predict output correctly for any input to an arbitrary program.

$$\text{PSR} = \frac{1}{|\mathcal{P}|} \sum_{p \in \mathcal{P}} \prod_{i=1}^{n} \mathbb{1}(f(p, x_{p,i}) = y_{p,i}) \quad (1)$$

- **Robust Drop ($\mathcal{R}_\Delta$)**: The absolute performance degradation when moving from canonical to perturbed inputs. The greater the magnitude of the drop, the less robust the model's performance.

$$\mathcal{R}_\Delta = \text{Accuracy}_{C_{\tilde{o}}} - \text{Accuracy}_{C_o} \quad (2)$$

3 Methodology

This section presents our methodology, starting with our experimental design for each RQ, and closing with our model selection.

3.1 Experimental Design

We now detail our experimental design for each of our RQ.

RQ1. We perturb CruxEval inputs using a type-aware mutation algorithm following Liu et al. [28]. We produce perturbed inputs using the original input & program pairs in CruxEval, using rejection sampling. We mutate the original CruxEval input, reject the input if it is already in the set of inputs for that particular program, and otherwise execute the code with that input, record the output, and save samples that raise exceptions separately in a new CruxEval$_{\text{exc}}$ dataset. If an exception is raised, we record its type and message *e.g.*, `IndexError` and '`list index out of range`'.

As CruxEval programs are small and simple, we enforce a 5 second execution time limit, which when exceeded raises a custom `TimeoutError` exception, suggesting an infinite loop or recursion[1]. We limit the number of distinct perturbed inputs to 10 due to experimental (LLM usage) costs. Some programs have fewer than 10 inputs due to implicit input constraints *e.g.*, a program may enforce acceptable values for some argument x with a cardinality < 10. We exclude such programs from our study to ensure equal sample sizes, and leave constraint-aware perturbation to future work.

The ability to predict that an exception will be raised while executing a given program-input pair $(p, x_{p,i})$, and specifically *which* exception, is helpful while debugging & maintaining code. CruxEval ignores this issue; we, however, also gather & include input samples that raise runtime exceptions in our enhanced CruxEval$_{\text{exc}}$, thus extending output prediction to cover exceptions.

For exception prediction, we initially labeled a response as correct if the ground-truth exception type or message was present anywhere in the model generation. Upon reviewing model generations, we found exceptions that were correctly predicted, but did not contain the exception type verbatim nor the exact message *e.g.*, for ValueError: '`This code will crash because arguments must have the same length`'. To minimize false negatives, we subsequently devised heuristics for TimeoutError, ValueError, TypeError, and IndexError. For each, we define 1-5 substrings that indicate a correct prediction *e.g.*, '`infinite`' for TimeoutError.

We utilize two notions of correctness: 1) Any Match: *any* exception *type* or any heuristic substring is present in the model's response, regardless of what exception is recorded during execution (ground truth). This captures the ability to predict that *some* exception will occur. 2) Strict Match: The ground-truth exception type or message is present in the model's response, or a heuristic

[1] Gu et al. [15] use a 3 second timeout, which we increase to 5 due to local compute resource constraints.

substring is present for that *particular* exception. In contrast to 1), this captures the ability to predict *which* exception will occur.

Our evaluation begins with the original CruxEval few-shot prompt Gu et al. [15], as shown in Figure 2. The answers are in the form `[ANSWER] assert $expr$ == $value$ [/ANSWER]`. We extract the predicted result `$value$` from model responses using heuristics and regular expressions, ensuring that left-hand side of the assertion matches the query ground truth. We compare the extracted output (right-hand side) with the gold-truth output recorded during execution using a Python expression evaluator[2], minimizing the effects of stylistic differences such as quotations and whitespace. We found that in some cases, models produced explanations, chain-of-thought, or further examples using the provided function, subsequent to the correct answer. To not penalize models in such cases, we label a response containing multiple assertions as correct, if *any* of the extracted assertions are correct, *i.e.*, both the left and right-hand side of the assertion match the ground truth.

We study multiple open-source models and three frontier models. We include base models, instruction-tuned models, and both models trained for general reasoning, and code reasoning.

RQ2. We perturb the original CruxEval$_o$ programs with meaning-preserving transformations (MPTs), producing CruxEval$_p$ with four groups detailed below as *Syntax Transform*. We additionally perturb CruxEval$_o$ with variable renaming to produce CruxEval$_v$, further detailed below as *Variable Renaming*. We execute the code with the original input and verify output equivalence as a sanity-check of semantic preservation.

We hypothesize that the combination of MPTs and input perturbations would create an antagonistic effect, further degrading LLM performance on the CruxEval output prediction task. To help characterize these dynamics, we isolate the individual contributions of each class of perturbations in this paper. This granular approach is a prerequisite for understanding the underlying failure modes before addressing the compounding complexities of their interactions, which we reserve for future inquiry.

Syntax Transform Following Chakraborty et al. [9], we apply OperandSwap, BlockSwap, ForWhileSwap, and DeadCodeInsertion transformations, producing CruxEval$_p$.

Variable Renaming We mine DyPyBench, a dataset of 50 large, popular Python projects [7], for identifiers and their occurrences. For each CruxEval program, we rename all identifiers with randomly sampled ones, weighted by number of occurrences, from the mined identifier distribution, *i.e.*, the most common identifiers are more likely to be sampled. This ensures that the identifiers are unlikely to be rare, and thus carry higher likelihood for the model, while also likely to be out-of-context. Thus, the renamed variables are generic, and unlikely to be relevant; we refer to this setting as CruxEval$_v$.

We repeat our model evaluation as described in Section 3.1.

RQ3. We employ both static analysis to identify programs with loop constructs, and dynamic analysis (*tracing*) to record execution behavior for a given input *e.g.*, executed line-numbers, number of loop iterations, *etc.* We focus on loops, *i.e.*, `for`/`while` and comprehensions. We record the number of decisions encountered in each loop *e.g.*, a `for` loop with one `if`, with a compound condition *e.g.*, `(x > 5) and (y < 3)` in the header (and no further conditions in its body), would have a total of **three** decisions: the iterator itself, and the two sub-expressions in the `if` condition. With two iterations, the execution encounters a total of *six* decisions.

[2] https://pypi.org/project/asteval/

3.2 Model Selection

To provide a view of how LLM architecture influences code understanding, we selected a diverse suite of 14 models. Our selection criteria prioritized model size, training regime, and access model (open-weights *vs.* proprietary). The resulting cohort includes: Qwen2.5 Math 1.5B and 7B [48], DeepSeek R1 Distilled 1.5B, 7B, 8B [11], Qwen2.5 Base and Instruct (7B and 14B) [33], NVIDIA OpenCodeReasoning Nemotron 7B and 14B [1], Llama 3.1 8B [14], GPT-5 Nano, GPT-5.2, and Gemini 3 Pro. We cover a parameter size range from 1.5B to 14B (proprietary model size unknown at time of writing), training regimes completion, instruction-following, reasoning, code-reasoning, and access model (3 proprietary, 11 open-weights). We also note that each of the DeepSeek distilled models has its non-reasoning ancestor.

To follow developer workflows using frontier models, which often neglect manual parameter adjustment [12], we accessed GPT-5.2, GPT-5 Nano, and Gemini via their default API configurations. To mitigate experimental bias, we performed each query independently to avoid context-leakage [16]. No explicit system prompts were utilized, ensuring the evaluation focused strictly on the models' baseline performance and inherent knowledge [29].

4 Results

We evaluate model performance across our three research questions, measuring robustness to input perturbation (RQ1), code perturbation (RQ2), and execution decisions (RQ3). The following sections report the findings of our experiments across our studied models.

4.1 RQ1

Table 1: Accuracy & robustness metrics for original *vs.* perturbed CruxEval *inputs*. Arrow ↑ indicates higher values are preferred. Each model was evaluated on 684 programs, for which the original benchmark input and 10 unique perturbed variants were available.

Model	C_o (↑)	$C_{\tilde{o}}$ (↑)	$C_{o \cup \tilde{o}}$ (↑)	PSR (↑)	$\mathcal{R}_\Delta$ (↑)
Qwen2.5 Math 1.5B	12.48	12.22	12.24	1.91	−0.26
DS R1 1.5B	38.45	37.50	37.59	4.39	−0.95
Qwen2.5 Math 7B	34.94	33.27	33.43	11.11	−1.67
Qwen2.5 Instr 7B	40.94	41.81	41.73	16.81	+0.88
Nemotron 7B	46.20	43.26	43.53	4.39	−2.94
DS R1 7B	65.06	62.62	62.84	27.34	−2.44
Llama 3.1 8B	36.40	33.82	34.05	11.84	−2.59
DS R1 8B	64.91	62.91	63.09	29.68	−2.00
Qwen2.5 14B	22.08	23.20	23.10	3.80	+1.13
Qwen2.5 Instr 14B	47.21	47.84	47.79	21.11	+0.63
Nemotron 14B	67.25	65.06	65.26	33.48	−2.19
GPT-5 Nano	96.78	96.61	96.62	82.75	−0.18
GPT-5.2	**99.12**	79.61	81.38	56.14	−19.52
Gemini 3 Pro	99.10	**99.36**	**99.34**	**97.16**	**+0.25**

In Table 1, we observe that accuracy ranges widely across models, from as little as 12.48%, to as high as 99.12%. Our results show the effectiveness of reasoning post-training *e.g.*, QWEN2.5-MATH-1.5B's accuracy jumps from 12.48% to 38.45% after the DeepSeek R1 distilled finetune. Our best open-source model reaches 67.25% accuracy, far lower than the best-performing frontier model GPT-5.2's accuracy of 99.12%.

Considering $\mathcal{R}_\Delta$, we observe four models that achieved higher accuracy on the perturbed inputs than the original input, resulting in positive $\mathcal{R}_\Delta$ values. On the other hand, the majority of open-weight models did worse on the perturbed inputs, with an average $\mathcal{R}_\Delta$ of -1.127. In contrast, the frontier model GPT-5.2 suffered a significant $\mathcal{R}_\Delta$ of -19.52.

We also report (PSR) numbers, the proportion of programs for which models predict *all outputs correctly*. These are much lower, reaching only 56% for GPT-5.2. Arguably, correct performance for *several inputs for the same program* is a better measure of how well a model actually understands a program; if a model truly understands a program, it should get the output correct on all inputs! The drop for GPT-5.2 from 99% overall correct *for all inputs* to just 56% correct *for all programs*, suggests that its output prediction capability for a random program on *any* given input is much less reliable than its 99% performance on the original CRUXEVAL might indicate. We explore this concerning issue further in Section 5.

A surprising performance gap exists between GPT-5 Nano and GPT-5.2. While GPT-5 Nano is marketed as a smaller, more efficient version of GPT-5 released in August 2025, and GPT-5.2 is marketed as a flagship model released in December 2025, GPT-5 Nano outperforms GPT-5.2 on $C_{\tilde{o}}$ by 17%, $C_{o \cup \tilde{o}}$ by 15.24%, and PSR by 26.61%. Furthermore, its $\mathcal{R}_\Delta$ of -0.18 is greater than the $\mathcal{R}_\Delta$ of -19.52 for GPT-5.2. These results suggest that GPT-5 Nano is more robust than its nominally stronger counterpart. We hypothesize that it is a quantized, and possibly distilled, version of GPT-5. Prior work [4, 6, 23] suggests quantization exerts a regularizing effect, steering optimization toward flatter minima that exhibit greater robustness to perturbation, and a reduced tendency for overfitting. If our hypothesis holds, quantization-regularization may help explain the greater generalization we observe for GPT-5 Nano.

Notably, Gemini 3 Pro exhibits very high performance and impressive robustness: its $\mathcal{R}_\Delta$ is small and *positive*, and its PSR on the perturbed datasets is not that much lower than its performance on original CRUXEVAL dataset.

We subsequently investigated, for the best-performing frontier models, the LLM's ability to reason about input perturbations that result in runtime exceptions. In Table 2, we report the frontier model performances on $\text{CRUXEVAL}_{\text{exc}}$ using the canonical CRUXEVAL prompt, for both correctness criteria. In general, we observe that models find it more difficult to predict exceptions. Table 2 shows the accuracy of our 3 best-performing frontier models on predicting exceptions; the first column is a strict match, evaluating if the model can predict precisely which exception is thrown; the second column is "Any" which credits the model when any exception is predicted. It's noteworthy that models actually manage to predict the *right* exception in most cases that they manage to correctly predict that *some* exception is thrown. In Section 5, we explore the type of exceptions that are strictly correctly predicted, and then delve into the factors potentially related to poor exception performance, including the potentially prompt-compliant ("sychophantic") tendency of instruction-tuned LLMs to produce a plausible, fake, non-erroneous output, even when the actual output is an error.

Table 2: Performance of the frontier models on accuracy of prediction the precise exception that actually occurs ("Strict Match") or that some exception occurs ("Any Match")

Model	Strict Match %	Any Match %
GPT-5 Nano	**71.95**	**73.16**
GPT-5.2	14.77	14.77
Gemini 3 Pro	48.56	50.72

> **Answer for RQ1:** We find that most models perform worse with the perturbed inputs (average $\mathcal{R}_\Delta$ of -1.1), with GPT-5.2 being the most affected ($\mathcal{R}_\Delta$ of -19.5); furthermore, we notice a big performance degrade on predicting exceptions; we explore this further in Section 5.

4.2 RQ2

In Table 3, we observe generally minor differences when shifting from the original CRUXEVAL programs to their transformed (with Meaning-Preserving Transformation, "MPT") and Renamed variants. In contrast to Section 4.1, we only perturb the program *code* in this setting, as opposed to the *inputs i.e.*, test cases.

Similar to our observations in Table 1, the reasoning distilled models show remarkable gains over their base counterparts. For instance, Llama 3.1 8B accuracy on CRUXEVAL jumps from 37.23% to 67.82% in its DeepSeek R1 Distill ("DS R1 8B" in Table 3) version. However, these models also degrade under perturbation, with DeepSeek R1 Distill 8B dropping to 61.66% on MPT (-6.16 Δ) and

Table 3: Average model accuracy on canonical CRUXEVAL, code-perturbed CRUXEVAL_p, and CRUXEVAL_v. Asterisks *, **, and *** represent McNemar test $p < 0.05$, 0.01, and 0.001 respectively, for original *vs.* code-perturbed. As each model is a different experiment, we do not make claims across all models based the *p*-values and thus do not apply corrections.

Model	CRUXEVAL	CRUXEVAL_p	CRUXEVAL_v
Qwen2.5 Math 1.5B	12.25	9.27*	11.13
DS R1 1.5B	40.00	40.45	35.50*
Qwen2.5 Math 7B	37.00	35.53	34.38
Qwen2.5 Instr 7B	42.63	42.98	40.50
Nemotron 7B	47.00	37.22***	44.75
DS R1 7B	62.63	60.25	61.50
Llama 3.1 8B	37.88	34.55	34.63*
DS R1 8B	66.13	61.66	61.25**
Qwen2.5 14B	22.50	27.53	16.00***
Qwen2.5 Instr 14B	48.88	45.65	47.00
Nemotron 14B	68.63	57.87***	66.13
GPT-5 Nano	97.20	87.92***	93.88***
GPT-5.2	**99.24**	76.40***	75.75***
Gemini 3 Pro	**99.24**	**95.79*****	**98.37**

61.25% (−6.57 Δ) on Renamed. Among these open-source models, DeepSeek R1 Distill 14B emerges as the strongest performer, achieving 76.75% on the original programs, and maintaining high scores across both perturbation settings. Unexpectedly, the 14B model outperforms its 32B variant. We observe a surprisingly narrow performance gap between R1 14B and GPT-5.2.

The impact of variable renaming and MPTs is particularly evident for the frontier model GPT-5.2, showing a sharp decline from a near-perfect 99.24% to around 76.40% for both MPT and Renamed. In contrast, the decline for Gemini 3 Pro is much smaller, dropping from 99.24% to 95.79% and 98.37% for MPT and Renamed, respectively. On our dataset, despite matching Gemini-3-Pro's performance on the original CruxEval, Gpt-5.2 is more sensitive to surface-level syntactic code changes than Gemini-3-Pro. This is surprising as GPT-5.2 is near-perfect on the original benchmark, is newer (Jan 2025 *vs.* Dec 2025), and likely larger in terms of parameter count. GPT-5 Nano, while also being a strong performer on the original code, is significantly impacted by MPT and Renaming although not as much as GPT-5.2 *per se*.

> **Answer for RQ2:** We find that in most cases, the performance decreases somewhat for CruxEval$_p$, and even increases in a couple of cases (DeepSeek R1 1.5B, and Qwen 2.5 14B). For CruxEval$_v$ (variable renaming), performance does consistently drop (average $\mathcal{R}_\Delta$ of −4.3), but the differences are surprisingly low. The most noteworthy change is that one of the best performing models, GPT-5.2, suffers a dramatic performance drop (≈−23) on both renaming and transformation; the other frontier model, however, remains more robust.

4.3 RQ3

We examine the results of decisions encountered during execution. In Figure 1, we visualize the results from Table 4. We investigate the relationship between the number of decisions encountered on an execution path for a given program-input pair $(p, x_{p,i})$, and model accuracy. Each box in the figure represents the 14 model accuracies across n samples of output predictions for program-input pairs.

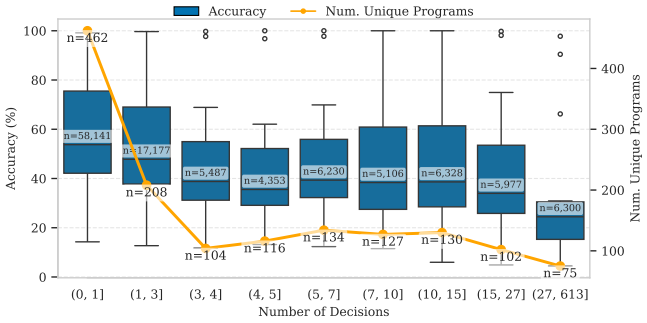

Figure 1: Number of decisions encountered on execution path *vs.* model accuracy on output prediction. The yellow line shows the varying number of sample programs with increasing numbers of conditions on the execution paths

Table 4: Binned number of decisions encountered during execution *vs.* mean model accuracy. Columns report number of samples (model, program, input triples), unique programs and unique inputs per bin.

Number of Decisions	Samples	Accuracy (%)	Unique Programs	Unique Inputs
(0, 1]	58,141	58.54	462	4,154
(1, 3]	17,177	55.48	208	1,227
(3, 4]	5,487	47.49	104	392
(4, 5]	4,353	45.16	116	311
(5, 7]	6,230	48.06	134	445
(7, 10]	5,106	48.45	127	365
(10, 15]	6,328	48.42	130	452
(15, 27]	5,977	44.75	102	427
(27, 613]	6,300	33.65	75	450

Due to the long-tail distribution of decision counts that arises from certain program-input pairs having many loop iterations, we bin the decision counts into nine bins for sufficient sample sizes—one for no decisions encountered (*e.g.*, straight line code), and eight *quantile* bins, with roughly comparable sample sizes varying from 4,353 to 17,177 observations.

Additionally, we plot the number of unique programs in each bin on the second (right) y-axis. Given that the sum of unique programs (1,458) exceeds the number of programs in CruxEval, it follows that execution paths and thus decision counts vary between inputs for the same program.

We observe a negative relationship between number of decisions and model output prediction correctness. A Mann-Whitney U test on pooled model results indicated that the number of decisions encountered was significantly lower for correct predictions than for incorrect ones ($U = 1.436{\times}10^9$, $p < .001$). While the effect size as measured by the rank-biserial correlation was weak ($r_{rb} = -0.129$), this suggests that, as the number of decisions a model must reason about increases *e.g.*, an `if` statement in a `for` loop, its ability to correctly reason about the code execution decreases. Further analysis per model found a minimum r_{rb} of −0.314 (Nemotron 14B), a median of −0.167, and maximum of 0.273 (Gemini 3 Pro). As the only model with a positive (surprisingly strong) effect, Gemini may be less hindered by the number of decisions.

Despite the overall negative trend in Figure 1, we note an increase in accuracy in the (5, 15] interval, where accuracy seems to recover somewhat from that in (3, 5]. The samples spike at this interval; further examination of this phenomenon and the Gemini 3 Pro outlier is left to future work.

> **Answer for RQ3:** We find that as the number of decisions increases to 3, there is a significant drop off in prediction accuracy; models generally struggle to predict outcomes as the number decisions encountered increases; however, our data suggests that Gemini 3 Pro's prediction accuracy in particular, does not decrease as decisions encountered increases.

5 Discussion

We begin by noting the dramatically poor performance of frontier models in predicting outputs for CruxEval$_{\text{exc}}$ compared to the

non-exception-raising input set *e.g.*, GPT-5.2 accuracy drops from ~81% in Table 1, to ~15% for exception prediction in Table 2 (also reproduced in the "Original" line in Table 7).

Table 5 breaks down the accuracy of the models in predicting exceptions, based on the type of exception actually raised when running the code sample. Due to the small sample sizes for several exception types in CruxEval$_{exc}$, the zero and 100% accuracy results in Table 5 lack a reliable basis for comparison and are therefore excluded from this analysis.

The most commonly thrown exception is TypeError, with 358 occurrences; the least common is NameError, with just 3 occurrences. Each exception is a standard Python exception, with the exception of TimeoutError, which occurs when the code took more than 5 seconds (wall time) to execute (see § 3.1), suggesting an infinite loop. We manually confirmed that the 11 programs for which this occurred were indeed infinite `while` loops.

For each exception, Table 5 shows, for each model, when that particular exception is predicted precisely ("Strict match"), and when the model predicts that *some* exception is predicted ("Any Match"). A strict match may occur if one of the heuristic substrings for that *particular* exception is contained in the model output *e.g.*, '`infinite`' for the ground-truth exception `TimeoutError`. Unlike infinite recursions (RecursionError), infinite loops are not a runtime exception in Python; thus, predicting the non-standard TimeoutError verbatim is challenging. We found that all correct predictions here are due to '`infinite`' appearing in the model output, per the heuristic. We later find that with prompt variations, Gemini 3 Pro can indeed predict TimeoutError *verbatim* for up to 19% of cases.

In light of these two correctness criteria, we find that the performance gap is particularly evident in the 358 occurrences of TypeError exceptions: Gemini 3 Pro correctly predicts that TypeError exception *per se* occurs just ~28% of the time, while predicting *some* exception ~32% of the time. In most instances of TypeError where GPT-5 Nano predicted *some* exception, it also almost always correctly predicted the specific TypeError, as evidenced by the minimal difference of 2.8% between 'any' and 'strict' match, 58.8% and 56.0% respectively. Upon further examination, we found that GPT-5 Nano correctly predicted the *exact* exception message for 48.74% of TypeErrors.

Table 5: Distribution of exception types in CruxEval$_{exc}$, along with model performance using the original CruxEval prompt.

Exception	Count	Any Match % (↑)			Strict Match % (↑)		
		Nano[a]	5.2[b]	Gem3[c]	Nano[a]	5.2[b]	Gem3[c]
TypeError	358	58.82	5.32	31.56	56.02	5.32	27.65
ValueError	152	81.58	15.13	75.66	81.58	15.13	75.66
IndexError	138	95.49	24.64	62.32	95.49	24.64	61.59
Attribute-Error	73	94.44	1.37	26.03	94.44	1.37	26.03
KeyError	54	94.44	83.33	98.15	94.44	83.33	98.15
Timeout-Error	36	27.78	0.00	63.89	27.78	0.00	63.89
Recursion-Error	10	100.00	0.00	100.00	100.00	0.00	70.00
Lookup-Error	5	0.00	0.00	0.00	0.00	0.00	0.00
Zero-Division-Error	5	100.00	20.00	80.00	100.00	20.00	80.00
NameError	3	0.00	0.00	0.00	0.00	0.00	0.00

[a] GPT-5 Nano [b] GPT-5.2 [c] Gemini 3 Pro

Table 7: Results for frontier models on CruxEval$_{exc}$. *Strict Match* requires the generation to contain either the ground-truth exception type, exception message, or a heuristic substring. *Any Match* requires the generation to contain *any* exception type or heuristic substring *e.g.*, model output contains "infinite", but ground-truth exception is `TypeError`.

Model	Prompt	Strict Match %	Any Match %
GPT-5 Nano	Original	71.95	73.16
	Exceptions	88.49	93.65
	Type	73.50	74.70
	Exc. + Type	**89.93**	**95.92**
GPT-5.2	Original	14.77	14.77
	Exceptions	80.58	88.61
	Type	15.59	15.71
	Exc. + Type	**81.30**	**91.13**
Gemini 3 Pro	Original	48.56	50.72
	Exceptions	98.20	98.92
	Type	81.89	84.41
	Exc. + Type	**98.92**	**99.64**

Among the exceptions that occur more frequently, TypeError appears to be hardest one for all our frontier models[3]. In Python, TypeError functions as a catch-all, spanning, by convention, a diverse range of errors, such as argument count mismatches and non-iterable access. Python experts also use it as a broad, domain-extensible error type. This diversity likely accounts for both its prevalence in CruxEval$_{exc}$ and its relative prediction difficulty. Whether encountered statically or dynamically, this difficulty in predicting TypeErrors cannot be ignored: TypeErrors *will be* encountered when writing & debugging code; thus, it would be desirable to improve the ability to predict occurrences of this error. We begin by first discussing possible reasons of poor performance in predicting exceptions in general, and then turn more specifically to TypeErrors.

First, we note that the ~15% value in "Original" line for GPT-5.2, in Table 7 was produced using the actual, unmodified CruxEval prompt [15]. Notably, this original prompt doesn't mention exceptions, as you can see if you inspect the CruxEval prompt, labeled "Direct Output Prompt" in Figure 2. This lacuna may have caused the frontier LLMs to perform badly. Instruction-tuned models have been reported to show sycophancy [35], where they follow instructions literally & narrowly, ignoring other relevant context. For further clarity, we modified the CruxEval prompt, adding **just** the blue text in the "Direct Output Prompt Type-Strict + Exceptions" prompt in the lower part of Figure 2, to tell the frontier LLMs to consider exceptions. This text comprises both an instruction and adds a few-shot example.

[3] While performance for LookupError & NameError are worse, we have too few samples of them to draw any conclusions.

Figure 2: The prompts we used: the "Direct Output Prompt" is from the original CruxEval paper [15]. The lower prompt adds instructions for *both* type-tracking and exception-tracking. We also experimented with each separately

Direct Output Prompt

```
You are given a Python function and an assertion containing an input to the
function. Complete the assertion with a literal (no unsimplified expressions,
no function calls) containing the output when executing the provided code on
the given input, even if the function is incorrect or incomplete. Do NOT output
any extra information. Provide the full assertion with the correct output in
[ANSWER] and [/ANSWER] tags, following the examples.
[PYTHON] def f(n): return n assert f(17) == ?? [/PYTHON] [ANSWER] assert f(17)
== 17 [/ANSWER]
[PYTHON] def f(s): return s + "a" assert f("x9j") == ?? [/PYTHON] [ANSWER]
assert f("x9j") == "x9ja" [/ANSWER]
[PYTHON] {code} assert f( {input} ) == ?? [/PYTHON] [ANSWER]
```

Direct Output Prompt (Type-Strict + Exceptions)

```
You are given a Python function and an assertion containing an input to the
function. Complete the assertion with a literal (no unsimplified expressions,
no function calls) containing the output when executing the provided code on
the given input. STRICTLY ANALYZE both the TYPES and VALUES of every variable.
Consider how type-specific operations and implicit type conversions affect
the final result. If the code raises an exception, complete the assertion with
the exception type and message. Do NOT output any extra information. Provide
the full assertion with the correct output in [ANSWER] and [/ANSWER] tags,
following the examples.
[PYTHON] def f(n): return n assert f(17) == ?? [/PYTHON] [ANSWER] assert f(17)
== 17 [/ANSWER]
[PYTHON] def f(n, p): return n[p] assert f([1, 2, 3], 4) == ?? [/PYTHON]
[ANSWER] assert f([1, 2, 3], 4) == "IndexError: list index out of range"
[/ANSWER]
[PYTHON] {code} assert f( {input} ) == ?? [/PYTHON] [ANSWER]
```

The results, as shown on the "Exceptions" line in Table 7, improve dramatically from ~15% for GPT-5.2 to ~81%, which is in the range of what we note for non-exception-raising inputs; in fact, it is able predict the presence of *some* exception ~88% of the time. Performance also improves for Gemini, from ~49% to ~98%, and for GPT-5 Nano, from ~72% to ~88%.

We now focus on the high prevalence of Type-related exceptions in Table 5. We hypothesized that models might be too narrowly focused on predicting an output *value*, and ignoring *types*; this seems specially undesirable for a dynamically-typed language like Python, where the interpreter tracks *both types and values*. We therefore tried additional prompting **just** for type-tracking, shown in the "DOP Type-Strict + Exceptions" prompt in red text. We find that type-track prompting does improve performance, although never as much as exception-track prompting; it helped a lot with Gemini, but less so for the two GPT models ('Type" line, Table 7). Delving into the data, we find that the type-track prompting doesn't *specifically help improve TypeError identification a great deal*; in fact, for the GPT models, it helps ValueError identification more! Interestingly, the Exception track prompting helps find TypeErrors *much more* than Type-tracking, in all models.

We also tried combining type-track prompting with exception-track prompting: this is the "DOP Type-Strict + Exceptions" prompt including both the blue and red additions. As seen in the "Exc. + Type" line, their combination always works best, although the improvements over the "Exceptions"-only prompt are modest.

While the combination prompt improves performance *for all* the frontier models, it does raise the question as to whether this improvement is only manifest for the exception-raising cases: would the combination prompt actually help for normally-running inputs as well? To study this, we used the combination prompt on all our normally-running inputs (both the original *and* perturbed inputs). The results are in Table 8.

Table 8: Evaluating the combined Type & Exception Tracking prompt on the (original and perturbed) inputs that run without exceptions

Model	Prompt	C_o (↑)	$C_{\tilde{o}}$ (↑)	$C_{o \cup \tilde{o}}$ (↑)	PSR (↑)
GPT-5 Nano	Original	96.78	96.61	96.62	82.75
GPT-5.2	Original	**99.12**	79.61	81.38	56.14
Gemini 3 Pro	Original	99.10	**99.36**	**99.34**	**97.16**
GPT-5 Nano	Exc.+Type	97.08	96.87	96.89	83.63
GPT-5.2	Exc.+Type	76.17	75.98	76.00	50.29
Gemini 3 Pro	Exc.+Type	**99.42**	**99.33**	**99.34**	**97.22**

For both GPT-5 Nano and Gemini 3 Pro, when we compare the change from "Original" prompt to the combined "Exc. + Type" prompt, we see improved performance for the original (C_o), the perturbed ($C_{\tilde{o}}$) and both together ($C_{o \cup \tilde{o}}$); we also see improved performance on the PSR score (the fraction of programs where every prediction is correct). Not surprisingly, improvements are larger for GPT-5 Nano than for Gemini 3, which already performed almost perfectly with the Original prompt. We do notice a substantial performance drop-off in GPT-5.2 when we use the combination prompt, from ~81% to ~76%. To investigate this 5% drop in accuracy, we inspected the results to check if GPT 5-2 is sycophantically predicting more exceptions in response to the combination prompt. We found that in about ~4% of the cases where it originally was correctly predicting the right values, it was *indeed* erroneously predicting exceptions. The causes for the rest of the errors remain unclear. As noted earlier in Section 4.1, we hypothesize the worse behavior of GPT-5.2 relative to GPT-5 Nano may be due to the latter being better regularized.

5.1 Threats to Validity

Internal Threats. Performance was highly dependent on instruction; for example, GPT-5.2 improved from 15.00% to 81.00% when prompted to track exceptions. Instruction-tuned models may follow prompts too literally, often failing to predict exceptions because the original prompt did not mention them. Both examples illustrate how the exact choice of prompt can significantly affect outcomes due to *prompt sensitivity*. Additionally, the choice of *model configuration* and sampling hyperparameters may affect results. While our usage of type-aware mutations aims to create a dense neighborhood of local inputs for each program, the approach may introduce latent *distributional biases* in perturbed inputs *e.g.*, longer input sequences, that have a causal relationship with model outcomes. Categorical accuracy for rare exceptions, such as *NameError* or *LookupError*, is unreliable due to *sample sparsity*. Finally, the use of regular expressions, heuristics, and their specific selection & implementations for extracting model responses may introduce *measurement error* or misinterpret the model's intended output.

External Threats. While CruxEval is currently the standard academic benchmark for code-reasoning output prediction, it consists of small, synthetically generated, standalone Python functions that

are a coarse-grained proxy for program execution, potentially masking the nuances of state management in larger systems. Thus, to reduce *benchmark bias*, our findings could be strengthened by applying our approach to multiple, diverse, real-world datasets in future work. By focusing our study on Python due to the choice of benchmark, we acknowledge a *language bias*. Challenges such as predicting *TypeErrors* may not apply to statically-typed languages, and model accuracy & robustness might differ on other languages under the same perturbations. The use of proprietary, black-box models such as GPT-5.2 or Gemini 3 Pro is tied to specific API versions and may change over time under *model decay*. Our findings are constrained by our perturbation methodology. Without a comprehensive study of perturbations, especially with respect to MPTs, our results may not necessarily generalize to other transformations.

6 Related Work

Robustness and Model Stability on Code. Early research evaluated code models against semantic-preserving transformations. Henkel et al. [19] and Dong et al. [13] used metamorphic testing (MPT) and "litmus" transformations to assess and augment model stability on code-captioning and classification tasks, respectively. ReCode [41] studied robustness in code *generation* under docstring and code perturbation. Wei et al. [43] applied coverage-guided fuzzing to study the reliability of an LLM determining the semantic equivalence of programs. While benchmarks like EvalPlus [28] provide comprehensive evaluation for generation, our work focuses on the robustness of understanding *given* code.

Recent works have studied deeper semantic perturbations: Hooda et al. [20] used counterfactual mutations, such as flipping branches, to test conceptual understanding, finding significant sensitivity to control-flow changes. Others have examined specific contexts such as poor readability [21], syntactic adversarial attacks in translation [50], and prompt instability in vulnerability detection [17].

Lam et al. [26] study code reasoning robustness only under code perturbation *e.g.*, insertion of misleading natural language comments. Our RQ2 is a partial reproduction of their core results; our RQ2 findings broadly align with and reinforce theirs. We generalize the application of perturbation to inputs. *Input perturbation* gives rise to both additional valid, and invalid, inputs (which may cause exceptions), allowing us to evaluate dependency on control flow decisions; our RQ1 and RQ3 explore these issues in detail.

Broader exploration of these topics, can be found in surveys by Asgari et al. [3] on metamorphic testing for code models, and Song et al. [36] on the robustness and reasoning failures of LLMs.

Benchmarking Code Reasoning and Execution. A significant body of work now benchmarks the deeper code-reasoning capabilities of LLMs. We refer to Ceka et al. [8] for a recent survey taxonomizing code reasoning techniques. Broad multi-task suites like CodeMMLU [30] and SX-Bench [47] assess general software principles, while LiveCodeBench-Exec [22], like CruxEval, assesses output prediction ability. CruxEval-X [46] and CodeSense [34] further extend these evaluations to multilingual and real-world repository settings. In contrast, REval [10] uses code generation benchmark data to assesses prediction of intermediate states during program execution, and model logical reasoning consistency on code-intelligence tasks of increasing difficulty. Finally, specialized tools such as ExeRScope [27] isolate runtime behavior and dynamic properties.

Prenner and Robbes [32] contribute a benchmark that evaluates models *specifically focused* on their ability to predict exceptions in failing programs, with a prompt tuned to find exceptions. We study robustness broadly on the output *and* exception prediction task on programs in CruxEval, finding limitations, notably for Type Exceptions, and explore prompt engineering to address these problems.

Patel et al. [31] discuss exception prediction by first building a CFG and then instructing an LLM use the CFG to predict execution results. We study the robustness of LLM code understanding in a setting wherein it lacks the support of a separate CFG analyzer. Bieber et al. [5] contribute a benchmark that evaluates LLMs' ability to statically determine which exceptions *might* be thrown by a program on *any* input, thus a more difficult task; robustness to perturbation was considered in their work.

Beyond execution tracing, research has moved toward functional equivalence and static analysis. While Core [45] evaluates static information flow, EquiBench [42] and ProbeGen [2] test whether models can determine if two programs are semantically identical. Notably, ProbeGen utilizes the model itself to generate inputs that disprove equivalence, enabling semantic clustering. Finally, recent work has addressed model confidence through calibration frameworks [37] to evaluate and improve code reasoning confidence [40].

7 Conclusion

Do language models robustly understand the meaning of programs? We explore this question using perturbations of the CruxEval benchmark, which requires models to correctly predict outputs for given program-input pairs. We perturb the *inputs* in CruxEval using type-aware mutations, and the *programs* using meaning-preserving transforms. Some of the perturbed inputs cause programs to throw exceptions, and we add these to our evaluation.

We find that performance on perturbed inputs and perturbed programs is generally a bit lower, notably *substantially* lower for the frontier GPT-5.2 model. We also find that performance on exceptions using the given prompt in CruxEval is also substantially worse, and report some prompting interventions that provide evidence of improvement. Finally, we also find that these exception-related interventions help all models improve performance on the original benchmarks, but actually *worsen* the performance of GPT-5.2.

Our findings support the conclusion that even frontier models lack robustness on the output prediction task. Future research could explore the causes of this brittleness.

References

[1] Wasi Uddin Ahmad, Sean Narenthiran, et al. 2025. OpenCodeReasoning: Advancing Data Distillation for Competitive Coding. arXiv:2504.01943 [cs] doi:10.48550/arXiv.2504.01943

[2] Miltiadis Allamanis and Pengcheng Yin. 2025. Disproving Program Equivalence with LLMs. arXiv:2502.18473 [cs] doi:10.48550/arXiv.2502.18473

[3] Ali Asgari, Milan de Koning, et al. 2025. Metamorphic Testing of Deep Code Models: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3766552

[4] MohammadHossein AskariHemmat, Reyhane Askari Hemmat, et al. 2022. QReg: On Regularization Effects of Quantization. arXiv:2206.12372 [cs] doi:10.48550/arXiv.2206.12372

[5] David Bieber, Rishab Goel, et al. 2022. Static Prediction of Runtime Errors by Learning to Execute Programs with External Resource Descriptions. arXiv:2203.03771 [cs.LG] https://arxiv.org/abs/2203.03771

[6] Aymen Bouguerra, Daniel Montoya, et al. 2026. Less Precise Can Be More Reliable: A Systematic Evaluation of Quantization's Impact on CLIP Beyond Accuracy. arXiv:2509.21173 [cs] doi:10.48550/arXiv.2509.21173

[7] Islem Bouzenia, Bajaj Piyush Krishan, et al. 2024. DyPyBench: A Benchmark of Executable Python Software. *Proc. ACM Softw. Eng.* 1, FSE, Article 16 (July 2024), 21 pages. doi:10.1145/3643742

[8] Ira Ceka, Saurabh Pujar, et al. 2025. How Does LLM Reasoning Work for Code? A Survey and a Call to Action. arXiv:2506.13932 [cs] doi:10.48550/arXiv.2506.13932

[9] Saikat Chakraborty, Toufique Ahmed, et al. 2022. NatGen: Generative Pre-Training by "Naturalizing" Source Code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 18–30. doi:10.1145/3540250.3549162

[10] Junkai Chen, Zhiyuan Pan, et al. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, 1869–1881.

[11] DeepSeek-AI, Daya Guo, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *Nature* 645, 8081 (Sept. 2025), 633–638. doi:10.1038/s41586-025-09422-z arXiv:2501.12948 [cs].

[12] Benedetta Donato, Leonardo Mariani, et al. 2025. Studying How Configurations Impact Code Generation in LLMs: The Case of ChatGPT. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 442–453. doi:10.1109/ICPC66645.2025.00055

[13] Zeming Dong, Qiang Hu, et al. 2023. MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 379–390. doi:10.1109/SANER56733.2023.00043

[14] Aaron Grattafiori, Abhimanyu Dubey, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs] doi:10.48550/arXiv.2407.21783

[15] Alex Gu, Baptiste Roziere, et al. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, 16568–16621.

[16] Chenchen Gu, Xiang Lisa Li, et al. 2025. Auditing Prompt Caching in Language Model APIs. In *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 20477–20496.

[17] Shuo Han, Tao Tan, et al. 2025. Prompting Instability: An Empirical Study of LLM Robustness in Code Vulnerability Detection. In *AI 2025: Advances in Artificial Intelligence*, Miaomiao Liu, Xin Yu, et al. (Eds.). Springer Nature, Singapore, 233–245. doi:10.1007/978-981-95-4969-6_18

[18] Dan Hendrycks, Collin Burns, et al. 2020. Measuring Massive Multitask Language Understanding. In *International Conference on Learning Representations*.

[19] Jordan Henkel, Goutham Ramakrishnan, et al. 2022. Semantic Robustness of Models of Source Code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 526–537. doi:10.1109/SANER53432.2022.00070

[20] Ashish Hooda, Mihai Christodorescu, et al. 2024. Do Large Code Models Understand Programming Concepts? Counterfactual Analysis for Code Predicates. In *Proceedings of the 41st International Conference on Machine Learning*. PMLR, 18738–18748.

[21] Chao Hu, Yitian Chai, et al. 2024. How Effectively Do Code Language Models Understand Poor-Readability Code?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*. Association for Computing Machinery, New York, NY, USA, 795–806. doi:10.1145/3691620.3695072

[22] Naman Jain, Alex Gu, et al. 2025. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *International Conference on Representation Learning* 2025 (May 2025), 58791–58831.

[23] Saqib Javed, Hieu Le, et al. 2025. QT-DoG: Quantization-Aware Training for Domain Generalization. In *Proceedings of the 42nd International Conference on Machine Learning*. PMLR, 26981–27004.

[24] Juyong Jiang, Fan Wang, et al. 2025. A Survey on Large Language Models for Code Generation. *ACM Trans. Softw. Eng. Methodol.* (July 2025). doi:10.1145/3747588

[25] Chris F Kemerer. 1995. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering* 1, 1 (1995), 1–22.

[26] Man Ho Lam, Chaozheng Wang, et al. 2025. CodeCrash: Exposing LLM Fragility to Misleading Natural Language in Code Reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

[27] Changshu Liu and Reyhan Jabbarvand. 2025. A Tool for In-depth Analysis of Code Execution Reasoning of Large Language Models. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1178–1182.

[28] Jiawei Liu, Chunqiu Steven Xia, et al. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, 21558–21572.

[29] Norman Mu, Jonathan Lu, et al. 2024. A Closer Look at System Message Robustness. In *Neurips Safe Generative AI Workshop 2024*.

[30] Dung Nguyen, Thang Phan, et al. 2025. CodeMMLU: A Multi-Task Benchmark for Assessing Code Understanding & Reasoning Capabilities of CodeLLMs. *International Conference on Representation Learning* 2025 (May 2025), 2614–2672.

[31] Smit Patel, Aashish Yadavally, et al. 2025. Planning a Large Language Model for Static Detection of Runtime Errors in Code Snippets. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 639–639.

[32] Julian Aron Prenner and Romain Robbes. 2025. ThrowBench: Benchmarking LLMs by Predicting Runtime Exceptions. arXiv:2503.04241 [cs.SE] https://arxiv.org/abs/2503.04241

[33] Qwen, An Yang, et al. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs] doi:10.48550/arXiv.2412.15115

[34] Monoshi Kumar Roy, Simin Chen, et al. 2025. CodeSense: A Real-World Benchmark and Dataset for Code Semantic Reasoning. arXiv:2506.00750 [cs] doi:10.48550/arXiv.2506.00750

[35] Mrinank Sharma, Meg Tong, et al. 2023. Towards understanding sycophancy in language models. *arXiv preprint arXiv:2310.13548* (2023).

[36] Peiyang Song, Pengrui Han, et al. 2026. Large Language Model Reasoning Failures. *Transactions on Machine Learning Research* (2026).

[37] Claudio Spiess, David Gros, et al. 2024. Calibration and Correctness of Language Models for Code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 495–507. doi:10.1109/ICSE55347.2025.00040

[38] Weisong Sun, Yun Miao, et al. 2025. Source Code Summarization in the Era of Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. IEEE Press, 1882–1894.

[39] Alex Wang, Yada Pruksachatkun, et al. 2019. SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems. In *Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[40] Shufan Wang, Xing Hu, et al. 2025. Open the Oyster: Empirical Evaluation and Improvement of Code Reasoning Confidence in LLMs. arXiv:2511.02197 [cs] doi:10.48550/arXiv.2511.02197

[41] Shiqi Wang, Zheng Li, et al. 2023. ReCode: Robustness Evaluation of Code Generation Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Anna Rogers, Jordan Boyd-Graber, et al. (Eds.). Association for Computational Linguistics, Toronto, Canada, 13818–13843. doi:10.18653/v1/2023.acl-long.773

[42] Anjiang Wei, Jiannan Cao, et al. 2025. EquiBench: Benchmarking Large Language Models' Reasoning about Program Semantics via Equivalence Checking. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, Christos Christodoulopoulos, Tanmoy Chakraborty, et al. (Eds.). Association for Computational Linguistics, Suzhou, China, 33856–33869. doi:10.18653/v1/2025.emnlp-main.1718

[43] Moshi Wei, Yuchao Huang, et al. 2023. CoCoFuzzing: Testing Neural Code Models With Coverage-Guided Fuzzing. *IEEE Transactions on Reliability* 72, 3 (Sept. 2023), 1276–1289. doi:10.1109/TR.2022.3208239

[44] Xin Xia, Lingfeng Bao, et al. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[45] Danning Xie, Mingwei Zheng, et al. 2025. CoRe: Benchmarking LLMs Code Reasoning Capabilities through Static Analysis Tasks. arXiv:2507.05269 [cs] doi:10.48550/arXiv.2507.05269

[46] Ruiyang Xu, Jialun Cao, et al. 2025. CRUXEVAL-X: A Benchmark for Multilingual Code Reasoning, Understanding and Execution. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, et al. (Eds.). Association for Computational Linguistics, Vienna, Austria, 23762–23779. doi:10.18653/v1/2025.acl-long.1158

[47] Kaiwen Yan, Yuhang Chang, et al. 2025. STEPWISE-CODEX-Bench: Evaluating Complex Multi-Function Comprehension and Fine-Grained Execution Reasoning. arXiv:2508.05193 [cs] doi:10.48550/arXiv.2508.05193

[48] An Yang, Beichen Zhang, et al. 2024. Qwen2.5-Math Technical Report: Toward Mathematical Expert Model via Self-Improvement. arXiv:2409.12122 [cs] doi:10.

48550/arXiv.2409.12122

[49] Boyang Yang, Zijian Cai, et al. 2025. A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications. arXiv:2506.23749 [cs] doi:10.48550/arXiv.2506.23749

[50] Guang Yang, Yu Zhou, et al. 2025. Assessing and Improving Syntactic Adversarial Robustness of Pre-Trained Models for Code Translation. *Information and Software Technology* 181 (May 2025), 107699. doi:10.1016/j.infsof.2025.107699