– Independent Work Report Fall 2025 –

Beyond Surface Cues: Evaluating Semantic Reasoning in Large Language Models

Yash Thakkar
Department of Computer Science, Princeton University
Advisor: Professor Aarti Gupta

Abstract

Large language models (LLMs) are increasingly used to assist with software development tasks such as code summarization, debugging, and program understanding. These models have strong performance on benchmarks, but it is unclear the kinds of reasoning these models rely on and the extent to which they genuinely understand program semantics. This work evaluates the robustness of Qwen2.5-Coder-32B-Instruct under controlled transformations that remove semantic cues, with a focus on code summarization and error detection. Using a systematic evaluation pipeline, programs are modified through identifier obfuscation and semantic-preserving transformations, and separate datasets of validated buggy code are constructed to assess debugging performance. Experiments suggest that the model captures structural aspects of program logic beyond simple pattern matching. In contrast, error detection performance declines more sharply as transformations increase in complexity, where the model often recognizes that a program is incorrect, but fails to precisely localize or classify it. These results demonstrate that current LLMs reliably capture high-level program intent, but they fall short when deeper, multi-step reasoning of program behavior is required. This gap limits their reliability for debugging and defect analysis, and addressing them will require training that emphasizes semantically challenging and adversarial code.

1. Introduction

Large language models (LLMs) are commonly integrated into software development workflows, where they support documentation, onboarding, code review, debugging, and rapid prototyping [1]. One of the most popular use cases of LLMs is their ability to understand and summarize code, which involves taking a piece of code and generating a natural language description of its functionality [2]. The practical stakes are high because an incorrect summary or diagnosis can mislead developers, steer debugging in the wrong direction, and create downstream errors that are difficult to detect. As these models become more trusted for large-scale development tasks, it becomes important to understand what signals they rely on and what kinds of reasoning they can actually perform.

Despite the strong performance of modern LLMs on common benchmarks [3], it remains unclear what these models are learning because they largely operate as black boxes. Many codebases contain meaningful identifiers (such as comments, variables, and doc-strings) that provide strong hints about intent because they can reveal a program's purpose even before control flow and data transformations are analyzed. The model can also produce plausible summaries by pattern matching because it is trained on a massive training corpus [4], which raises concerns about LLMs' effectiveness in semantic understanding beyond surface-level cues. The goal of this paper is to evaluate semantic robustness in code understanding by systematically probing *Qwen2.5-Coder-32B-Instruct* model using controlled transformations on the MBPP-Pro dataset, a self-invoking code generation task that evaluates progressive reasoning and problem-solving capability in LLMs [5]. Two complementary stress tests involving code obfuscation and error detection are designed to deliberately remove or corrupt superficial cues, while preserving executable structure. The first experimental test obfuscates the identifiers (like names and comments) in a program with random alpha-numeric strings, and the second test injects realistic bugs in a program to test the model's ability to reason about debugging the incorrect program behavior. By identifying the model's strengths and weaknesses, this paper will guide the development or fine-tuning of future models to make them more robust and effective in complex and unconventional programming tasks. To develop controlled experiments, additional

LLMs are used: *Claude Sonnet 4.5* is used to remove identifiers and inject bugs, and *GPT-5.1* is used as an LLM-as-a-judge to evaluate Qwen's performance and robustness against the experimental tests.

2. Background and Related Work

Code summarization is defined as the task of generating a natural language description that accurately captures the semantics and intent of a given code snippet. Early work in this area relied primarily on static program analysis and information retrieval (IR) techniques [6]. These approaches used surface-level features or cues, mapping them to predefined templates or retrieving similar code–summary pairs from a corpus. While effective for well-structured conventional code, these earlier approaches to code summarization struggled to generalize beyond shallow lexical cues in semantically complex program logic.

The emergence of neural sequence-to-sequence models marked a significant shift towards learning-based approaches to code summarization because it allowed the models to capture longer-range dependencies between code tokens and natural language descriptions [7]. Recently, LLMs pre-trained on a massive corpus of source code and natural language have significantly improved performance on code-to-text tasks [2]. These newer models learn semantic knowledge during pre-training and generate fluent, context-aware summaries for various programming tasks. Developers are starting to rely on them heavily for any programming task [1], increasing their productivity in certain domains for programming compared to others.

Despite these advances, a fundamental question remains unresolved: does performance on code summarization benchmarks reflect semantic understanding of programs or reliance on correlations and surface-level patterns? Studies have suggested that LLMs could be succeeding because they exploit statistical regularities in training data instead of constructing robust representations of program behavior [4]. This has directed the research to evaluate the models on semantic reasoning and error sensitivity, both necessary for code interpretability and debugging.

To analyze semantic code performance in models, researchers have developed benchmarks on various aspects of program reasoning. The MBPP (Mostly Basic Python Problems) benchmark was introduced to evaluate models on small, self-contained Python programming tasks that require basic algorithmic reasoning [8]. Building on this foundation, Yu et. al. developed MBPP-Pro as an extension of the original dataset with increased difficulty and rigor by introducing self-invoking code generation to evaluate whether an LLM can effectively build upon its own logic [5]. This approach presents a base problem followed by a second complex variation, and requires the model to utilize the program from the primary problem to solve the more complex one.

This study is closely aligned with Yu et. al's work because it also evaluates large language models on the MBPP-Pro dataset. Their results show a 10-15% drop in Pass@1 performance for the self-invoking variants [5]. Pass@1, the most commonly used metric for code-generation models, measures the probability that a model produces a correct solution on its first attempt. The performance degradation demonstrates the difficulty of the MBPP-Pro variants because they require multi-step reasoning that builds on previously generated code to solve the subproblems. The MBPP-Pro dataset can be used to evaluate robustness in large language models to learn how and why models fail. This study evaluates *Qwen2.5-Coder-32B-Instruct* using the MBPP-Pro dataset, focusing on interpretability and semantic awareness.

3. Approach

3.1. Overview

To reiterate, this study investigates whether LLMs exhibit genuine understanding of code programs or is instead driven by reliance on superficial cues like identifier names, stylistic conventions, and common solution templates. To probe this question, two complementary experimental stress tests are designed to deliberately remove or corrupt such cues while preserving executable structure: code obfuscation and error detection.

Both tests operate on professionally curated programming problems from the MBPP-Pro benchmark which emphasizes functional correctness, semantic diversity, and robustness. Instead of evaluating the model solely through forward code generation, the direction of model reasoning is flipped: models are asked to infer intent from code or to diagnose faults in otherwise plausible implementations. This framing builds on recent work by Yu et al. [5], directly measuring semantic understanding instead of focusing on surface-level token overlap or pass@k metrics.

Across both tests, a controlled generation pipeline is adopted in which ground truth semantics and error labels are known by construction as well as LLM judge to evaluate Qwen's response and reason about its performance. This enables fine-grained, reproducible evaluation of model behavior under realistic conditions.

3.2. Task 1: Code Obfuscation

3.2.1. Motivation and Research Question. Meaningful identifier names provide strong semantic shortcuts for code understanding. For example, names such as `count`, `result`, or `is_valid` often encode substantial intent, enabling models to infer functionality without reasoning about control flow or data transformations. Prior work has shown that LLMs trained on large code corpora are highly sensitive to such lexical cues [9], raising questions about whether they internalize true program semantics or merely exploit naming or rely on coding templates. The central question addressed by this task is: can an LLM recover program intent purely from structure and operations, when all meaningful identifier names are removed?

3.2.2. Obfuscation Strategy. Identifiers that are typically defined by the programmer (variables, function names, class names, and parameters) are to be replaced with randomly-generated, semantically-meaningless strings. These identifiers are the typical lexical cues that models are suspected to utilize in their code comprehension, so the strings are replaced while preserving the code program's behavior. Of note, all semantic and syntactic structures like the code's control flow (loops, conditionals, etc.), data flow, operations, constants, and imports are preserved. This design ensures that the obfuscated code behaves the same as the original code, remaining fully functional and executable, but eliminating one of the strongest sources of semantic leakage and serving as a diagnostic tool to assess semantic robustness.

Now, given an obfuscated program, the Qwen model is tasked to produce a natural language description to infer the problem statement, which requires abstraction from low-level operations to high-level intent, and conduct reverse inference. The model has to interpret control flow patterns, recognize algorithmic structure independent of surface cues, and synthesize a semantic description. A model that performs well under the given transformation demonstrates genuine program understanding instead of relying on identifiers.

3.3. Task 2: Error Detection

3.3.1. Motivation and Research Question. While code generation benchmarks measure whether a model can produce correct programs, real-world software development often requires diagnosing why a program is incorrect. Error detection and debugging demand lower-level reasoning than surface pattern recognition. Models must identify faulty logic, reason about counterexamples, and articulate corrective actions. It is not just about detecting the error and fixing it, it is also about understanding *why* the code errored in the first place. The central question addressed by this task is: can an LLM accurately detect, localize, classify, and explain programming errors in syntactically valid code?

3.3.2. Bug Injection Strategy. Starting from correct MBPP-Pro reference solutions, a single realistic bug is introduced into each program using a generator model (Claude in our pipeline). These injected faults are syntactically valid, logical, and subtle because they are designed to mirror mistakes commonly made by human programmers. The bugs are categorized into seven distinct buckets: logical errors, off-by-one mistakes, edge-case failures, algorithmic flaws, boundary condition errors, type errors, and missing validation. Every injected bug must have a similar structure and readability to the reference solution, and the program must fail on at least one test case to ensure a high-fidelity "ground truth" where each error is fully labeled by type, location, and failure mode. Lastly, the code with injected bug must have valid Python syntax and remain executable.

3.3.3. Error Analysis Task. When presented with a potentially flawed program, the model must execute a comprehensive debugging sequence: it must first judge the code's correctness, then pinpoint the exact location of any faults, categorize them into our predefined categories, explain the underlying logical failure, and finally propose a corrective path. This multi-part formulation reflects a realistic debugging workflow and evaluation across several dimensions, ranging from simple detection to the high-level semantic reasoning required for technical explanation and code synthesis. An LLM-as-a-judge is employed to analyze Qwen's ability to debug in self-invoking code generation tasks.

3.4. Summary

Together, these two experimental tests form a concrete approach to stress-testing semantic understanding in large language models. Code obfuscation removes lexical shortcuts, and error detection introduces semantic disruptions to emphasize reverse reasoning. From implementation to intent or diagnosis, this approach provides a more holistic testing environment to assess model robustness and discover its limitations.

4. Implementation

This section describes the system used to execute the two stress tests introduced earlier: identifier-only obfuscation and error detection. The implementation is organized as two similar pipelines (one for each of the two experimental tests) that share a common dataset format, consistent file name, and a standardized "LLM-as-a-judge" evaluation. In both pipelines, intermediate artifacts are saved to support reproducibility and later analysis. Full prompt templates and additional implementation details are provided in Appendix A and B.

4.1. Dataset & Model

4.1.1. Dataset. All experiments are conducted on the MBPP-Pro dataset, a stronger variant of the original MBPP benchmark designed to evaluate progressive reasoning. There are 378 data instances, each containing an original ("raw") problem and a modified ("new") version [5]. Each MBPP-Pro dataset entry is stored as a JSON record (indexed by a numeric `id`) containing: `raw_problem`, `new_problem`, `raw_solution`, `new_solution`, and `test_code`. MBPP-Pro serves as the single source of truth across both pipelines because downstream stages only add fields.

4.1.2. Target Model. For both tasks, the system queries **Qwen2.5-Coder-32B-Instruct**, henceforth referred to as Qwen, as the target model due to its state-of-the-art performance among open-source large language models. Its sophisticated pre-training distribution is a mixture of 70% code, 20% natural language text, and 10% mathematical data totaling approximately 5.5 trillion tokens [10]. This specific data composition is designed to preserve general reasoning and mathematical logic alongside specialized programming knowledge. Qwen is uniquely equipped for both the evaluation tasks at hand, which require code synthesis paired with the high-level semantic reasoning needed to infer intent from structure and diagnose complex program faults.

4.1.3. Generator model. To produce controlled perturbations of the reference solutions, the system uses **Claude Sonnet 4.5** model in a dual-role capacity: (i) identifier-only refactoring to generate obfuscated code, and (ii) realistic bug injection for the error detection task. The key implementation constraint for both is that outputs must remain syntactically valid Python and preserve the overall structure to ensure the transformations remain "close" to the reference code. The synthetic data generation using Claude follows a well-defined rubric and structure for each of the experiments to ensure consistency and scalability across the entire MBPP-Pro dataset. Any problems in the pipeline related to the generator model is logged for debugging, and failed data entries are retried up to a pre-defined number. If the generator produces an inadequate response, the correspond data entry is discarded and not considered for evaluation.

4.1.4. Judge model and structured evaluation. For scalable scoring, the system uses **GPT-5.1** [11] as the LLM-as-a-judge. The judge is constrained to return JSON (scores + structured analysis) to mitigate hallucinations in formatting or inconsistent responses. This streamlines the evaluation and helps aggregate the performance of the Qwen model on the two tasks. For the code obfuscation task, the LLM judge has to compare Qwen's response with the ground-truth MBPP-Pro dataset, following specified pre-defined metrics to ensure consistency. If the LLM judge fails to follow the pre-defined template or runs into unaccounted errors, the data entry is discarded from evaluation.

4.2. Obfuscation Pipeline

The obfuscation pipeline instantiates the identifier-removal experimental test, and it consists of three stages with one output dataset per stage.

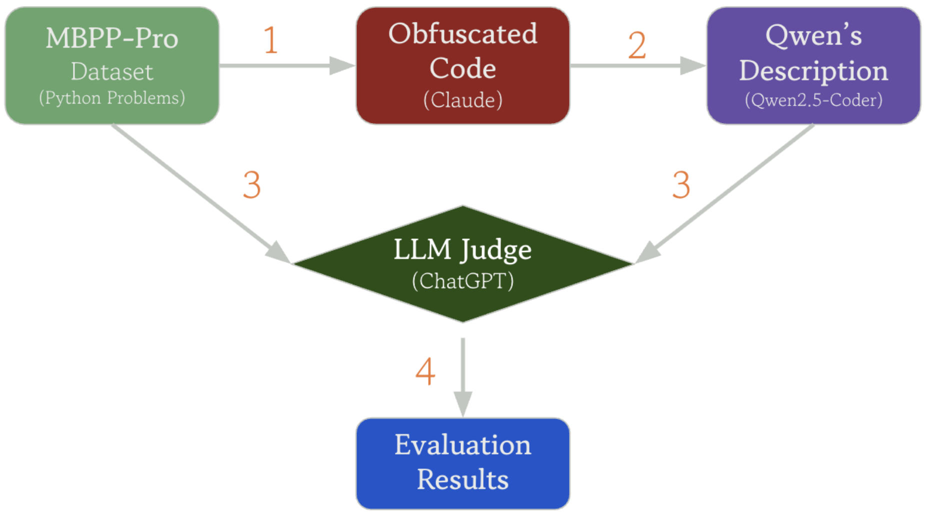
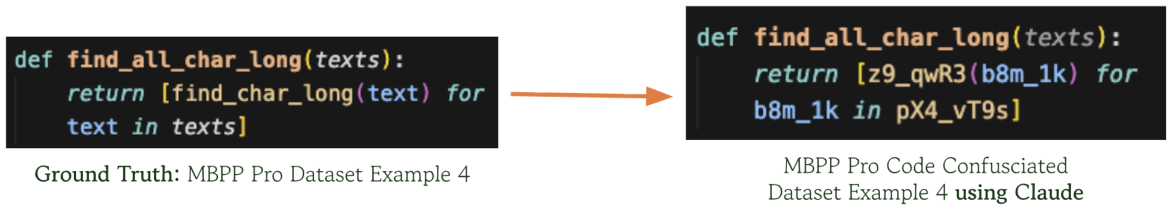


Figure 1: Evaluation Pipeline for Code Intent Inference. (1) Reference solutions from the MBPP-Pro dataset are refactored by Claude to generate obfuscated code. (2) Qwen2.5-Coder-32B-Instruct processes the obfuscated code to produce a description. (3) The LLM judge (GPT-5.1) compares the generated description against the ground truth. (4) Final evaluation results are aggregated via structured JSON.

4.2.1. Stage 1: Identifier Obfuscation. The goal is to produce an obfuscated variant of each data entry using *Claude Sonnet 4.5* to change the identifiers while preserving the program behavior and readability structure. The LLM generator is instructed to (i) rename all identifiers (functions, parameters, variables, classes), (ii) preserve control flow, general code structure, literals, and imports, and (iii) return a strict JSON object containing obfuscated `raw_solution` and `new_solution`. The transformation preserves the code structure and intended runtime behavior to ensure isolated testing on the impact of lexical cues on model's performance. The output is a new JSON file per dataset example containing the obfuscated code fields, saved under the directory `mbpp_pro_code_obfuscation`, mirroring the original dataset indexing.



```
def find_all_char_long(texts):
    return [find_char_long(text) for
    text in texts]
```

Ground Truth: MBPP Pro Dataset Example 4

```
def find_all_char_long(texts):
    return [z9_qwR3(b8m_1k) for
    b8m_1k in pX4_vT9s]
```

MBPP Pro Code Confusciated
Dataset Example 4 using Claude

Figure 2: An example of code obfuscation from the MBPP-Pro dataset. The left panel shows the original "Ground Truth" Python function, while the right panel shows the same code without identifiers by Claude, where variable and function names are replaced with randomized alphanumeric strings.

4.2.2. Stage 2: Qwen Intent Inference. To evaluate whether Qwen can infer the underlying task intent from code without identifiers, Qwen receives the obfuscated `raw_solution` and `new_solution` (as plain code blocks), and an instruction to produce concise natural-language problem descriptions each solution implements. Qwen is required to emit a strict JSON for each data entry, stored in a new directory called `qwen_code_obfuscation_description_response`, for later qualitative analysis.

4.2.3. LLM-as-a-Judge Scoring. Qwen's inferred descriptions will be scored against ground truth using a rubric (described in detail in the Evaluation Section) by the *GPT-5.1* model. The LLM-as-a-judge is given (i) ground-truth (MBPP-Pro dataset), (ii) Qwen's inferred descriptions on the obfuscated dataset, and (iii) the corresponding obfuscated code for context. All evaluations, including numeric scores and detailed analysis, are saved as structured JSON in the `llm_judge_evaluations` directory. The LLM judge follows a strict schema to ensure consistency and to prevent evaluation drift. These structured results are then compiled into a final `summary.json` that displays performance averages and score distributions for detailed evaluation.

4.3. Error Injection Pipeline

The error detection pipeline mirrors the obfuscation pipeline structurally, but operates on incorrect code variants and evaluates diagnostic reasoning. It also consists of three stages, with one output dataset per stage.

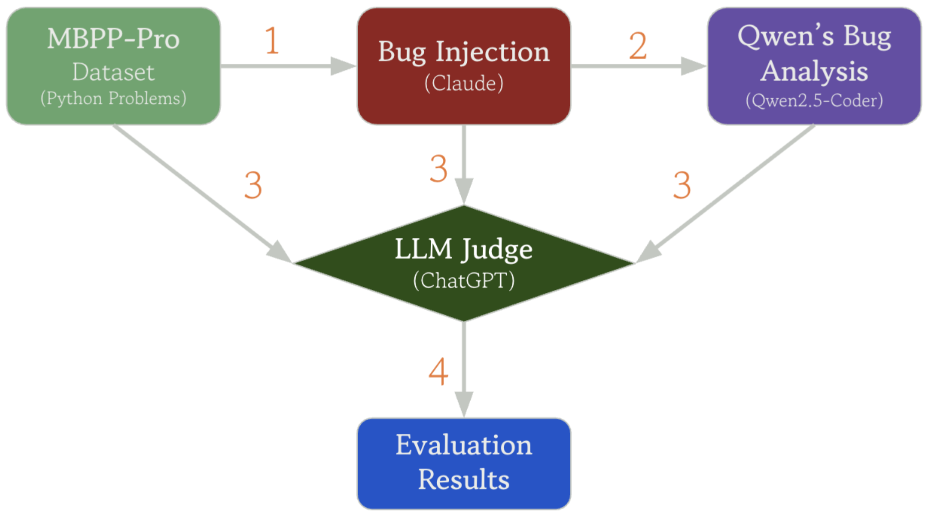


Figure 3: Evaluation Pipeline for Automated Bug Diagnosis. (1) Claude injects realistic logical faults into reference solutions from the MBPP-Pro dataset. (2) Qwen2.5-Coder-32B-Instruct analyzes the buggy code to locate, classify, and explain errors. (3) The LLM Judge evaluates the model's diagnosis against the ground-truth bug data and original solution. (4) Structured Evaluation Results are generated for final scoring.

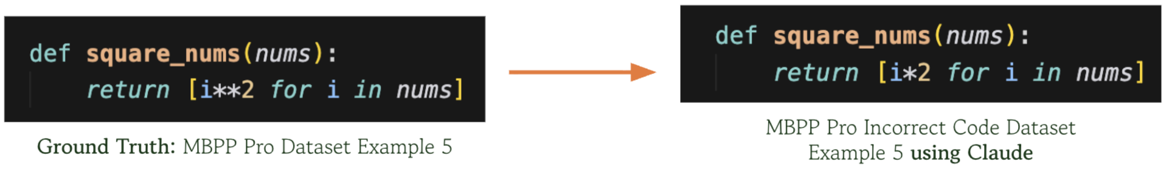
4.3.1. Stage 1: Bug Injection. The goal is to use **Claude Sonnet 4.5** to generate an incorrect program that remains syntactically valid, but fails at least one reference unit test. Each program is assigned a specific bug category (as reported in Table 1) based on a randomized seed to ensure reproducibility. For each injected bug, the generator emits structured metadata:

- **Error Type**: The specific category of the bug.
- **Description**: What was changed and why it is incorrect.
- **Location**: The exact region of the code that is faulty.
- **Expected Failures**: Specific inputs or conditions that should trigger an error.

Error Type	Description & Implementation Strategy
Logical Error	Introduce a logical error such as wrong conditional logic, incorrect mathematical operation, or flawed algorithm.
Off-by-One	Introduce an off-by-one error in indexing, counting, or loop boundaries.
Edge Case Failure	Introduce a bug that fails on edge cases like empty inputs, single elements, or boundary values.
Algorithm Error	Use a fundamentally different (and incorrect) algorithmic approach.
Boundary Condition	Introduce an error related to boundary conditions or limits.
Type Error	Introduce incorrect handling of data types or format conversions.
Missing Validation	Remove or incorrectly implement input validation or assumptions.

Table 1: Error types and their implementation strategies.

Following generation, each mutated program is validated by execution against the original test suite. A bug is considered valid only if the program remains syntactically correct and causes at least one test failure. Any instances that pass all test cases are either regenerated or excluded to ensure the pipeline strictly measures semantic debugging capabilities. All validated instances are stored in the `mbpp_pro_incorrect_code` directory for subsequent evaluation.



```python
def square_nums(nums):
    return [i**2 for i in nums]
```

```python
def square_nums(nums):
    return [i*2 for i in nums]
```

Ground Truth: MBPP Pro Dataset Example 5
MBPP Pro Incorrect Code Dataset Example 5 using Claude

Figure 4: An Example of Bug Injection. The model mutates the ground-truth code (left) by introducing a logical error, changing an exponentiation operator to a multiplication operator (right), to create a syntactically valid but functionally incorrect code.

4.3.2. Stage 2: Qwen Bug Analysis. To evaluate Qwen's ability to detect faults and provide actionable diagnostics, the model is provided with the problem description, the buggy implementation, and the associated unit tests. The model creates a JSON that includes a correctness verdict, error location, error type, and a proposed solution. It follows a pre-defined rubric to test the model's ability to analyze and classify a bug, as well as propose a fix. Every response undergoes strict schema validation to ensure reliable data aggregation, stored in the `qwen_incorrect_code_analysis` directory for evaluation.

4.3.3. Stage 3: LLM-as-a-Judge Scoring. To evaluate Qwen's error-detection performance, **GPT-5.1** acts as an LLM Judge, comparing the model's analysis against the ground-truth metadata. The judge reviews the buggy code, the Stage 1 error metadata, and Qwen's structured output to assess diagnostic accuracy across several predefined dimensions. This can simulate real-world debugging workflow to evaluate model performance and robustness. The results are returned in a strict JSON schema and archived in the `llm_judge_error_detection` directory. Finally, these individual results are aggregated into a `summary.json` file to calculate average scores and error distributions.

4.4. Evaluation Design & Metrics

Task-specific metrics capture semantic understanding, robustness to transformations, and diagnostic reasoning, evaluated through a structured LLM-as-a-judge protocol. These metrics provide a principled way to quantify model behavior under controlled perturbations and form the basis for the empirical results presented in the following section.

For each problem instance, the LLM Judge is provided with ground-truth descriptions, Qwen's outputs, and the relevant code context (either obfuscated or buggy). The judge returns numeric scores (from 1 to 5) with structured analysis, requiring strict JSON parsing, and supports consistent aggregation across hundreds of instances.

4.4.1. Metrics for Code Obfuscation. For the code obfuscation task, the judge evaluates the following five metrics, with each metric scored on a scale from 1 to 5, to determine whether the model can infer program intent from structure alone once meaningful identifier names have been removed.

1. **Semantic Accuracy (raw/new)** measures whether Qwen correctly captures the core intent and functional (I/O) behavior of the code. A high score indicates that the model's description aligns with the ground-truth problem specification in terms of inputs, outputs, and overall purpose, even when identifiers provide no semantic guidance.
2. **Completeness (raw/new)** assesses whether Qwen recognizes important constraints, corner cases, and requirements that are explicitly stated or implicitly enforced in the ground-truth problem description. A higher score means that the model response correctly identified the main idea of the task without omitting the relevant details.
3. **Transformation Understanding** evaluates whether Qwen correctly explains how the `raw_problem` is transformed into the `new_problem`, consistent with the paired-task design of MBPP-Pro. A high score suggests the model can identify semantic differences between the two experimental tasks and is able to articulate constraints or changes in the original problem.
4. **Robustness to Obfuscation** measures whether Qwen's inferred intent remains accurate despite the removal of identifier names. This metric evaluates whether the model was truly reliant on control flow, data transformations, and algorithmic structure instead of lexical cues, which then reflects the model's robustness to superficial code changes.
5. **Overall Score** is computed as the mean of the above rubric dimensions and serves as a summary on the code obfuscation task.

4.4.2. Metrics for Error Detection. For the error detection task, the judge evaluates Qwen across the following seven diagnostic metrics that align with practical debugging workflows, with each metric again scored on a scale from 1 to 5. These metrics are intended to capture not only whether the model detects errors, but also whether it understands and explains them accurately.

1. **Error Detection Accuracy** measures whether Qwen correctly identifies that the code is incorrect and detects the presence of a genuine fault. A high score indicates that the model reliably distinguishes between correct and incorrect implementations.

2. **Error Location Precision** evaluates whether Qwen localizes the fault to the correct region of the code or the specific statement responsible for that failure. A high score means the model is able to precisely identify the location and source of the bug.
3. **Error Type Classification** assesses whether Qwen correctly maps the detected bug to one of the predefined categories: `logical_error`, `off_by_one`, `edge_case_failure`, `algorithmic_error`, `boundary_condition`, `type_error`, or `missing_validation`. Correct classification reflects that the model can discern between different types of debugging problems that are subtle and difficult to find.
4. **Error Explanation Quality** measures the clarity and correctness of Qwen's explanation of why the error causes the program to fail. Strong explanations correctly link the faulty logic to specific failure cases or violated assumptions.
5. **Completeness** evaluates Qwen's ability to characterize the bug without omitting key aspects of the failure mechanism or hallucinating additional issues that are not present in the ground truth.
6. **Fix Suggestion Quality** assesses whether the proposed changes would plausibly repair the bug and restore correct behavior, emphasizing practical usefulness instead of superficial or incomplete fixes.
7. **Overall Score** is computed as the mean of the above rubric dimensions and serves as a summary of the error detection task.

4.5. Implementation Notes for Reproducibility

Across both pipelines, the implementation is intended to be reproducible and auditable:

- **Deterministic indexing**: each artifact file is keyed by the MBPP-Pro problem `id`, ensuring stable joins across directories.
- **Schema validation**: all model-facing stages enforce the JSON schema.
- **Isolated artifact generation**: each pipeline stage writes its outputs to a separate directory without mutating upstream data.

- **Seeded randomness**: randomized processes (e.g., error-type assignment during bug injection) use fixed random seeds.

5. Results & Evaluation

This section thoroughly evaluates **Qwen2.5-Coder-32B-Instruct** on the two experimental tests introduced earlier: (i) semantic inference from identifier-obfuscated code, and (ii) error detection and analysis of realistic bugs. All experiments are conducted using the MBPP-Pro dataset, which pairs a "raw" task with a related "new" task to emphasize progressive reasoning. Across both tasks, performance is measured using an LLM-as-a-judge protocol for scalable, rubric-based scores for semantic quality and diagnostic usefulness.

5.1. Results of Code Obfuscation

The obfuscation evaluation covers 378 MBPP-Pro problems, with 376 valid judge evaluations (2 failures due to pipeline errors). This provides broad coverage across problem types and difficulty.

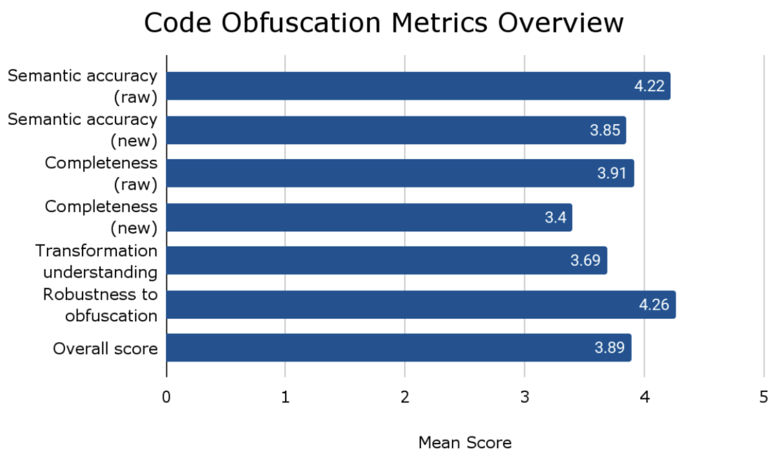


Figure 5: Code Obfuscation Metrics Overview. This horizontal bar chart displays the mean scores for seven performance categories on a scale of 0 to 5. Robustness to obfuscation achieved the highest mean score (4.26), followed closely by semantic accuracy (raw - 4.22). Conversely, while Completeness (new) received the lowest mean score (3.40).

5.1.1. Key Findings

Qwen is robust to identifier removal. Qwen demonstrates high resilience to obfuscation, achieving a Robustness score of 4.26. This suggests that the model does not merely "memorize" common function names but can successfully infer intent through structural semantics, specifically by tracing control flow, mathematical operations, and data transformations. This finding confirms that frontier code models have developed a deeper understanding than simply relying on lexical cues. Historically, LLMs were criticized for over-relying on semantic cues [9], but the results with Qwen suggest a significant increase in robustness. By maintaining high semantic accuracy and completeness even after total identifier obfuscation, the model proves it has shifted from simple pattern matching to functional abstraction, making the model resilient in unconventional or complicated programs.

Qwen consistently struggles more with explaining `new_problem`. There is an 8.9% decrease in Semantic Accuracy (dropping from 4.22 to 3.85) and a more significant 13.2% decrease in Completeness (3.91 to 3.40) from `raw_problem` to `new_problem`. This suggests that while the model can still correctly identify the core intent of the logic, it loses the ability to provide a comprehensive or detailed explanation when solving a complex ("new") problem that builds on top of the base ("raw") problem. This pattern is consistent with MBPP-Pro's design, where the "new" task typically introduces additional constraints or semantic shifts that require more precise abstraction to describe correctness. The model struggles to do multi-step reasoning and think in terms of systems with interconnected components.

Transformation understanding is a distinct failure mode. The transformation score (3.69) is lowest when compared to metrics (for "raw" problems), and it has relatively high variance (std. dev. 1.37). Qualitatively, Qwen often recognizes that *something changed*, but struggles to precisely articulate *what changed*. Even when the model understands the code's execution logic, it struggles with the high-level abstraction which is required to generalize the problem (e.g., expanding an input domain or adding post-processing steps). This suggests that self-invoking code generation robustness is weaker than single-task programming task.

5.2. Results of Error Detection

The error-detection evaluation spans the full MBPP-Pro dataset, with 378 total problems and 374 valid evaluations (4 excluded due to pipeline errors).

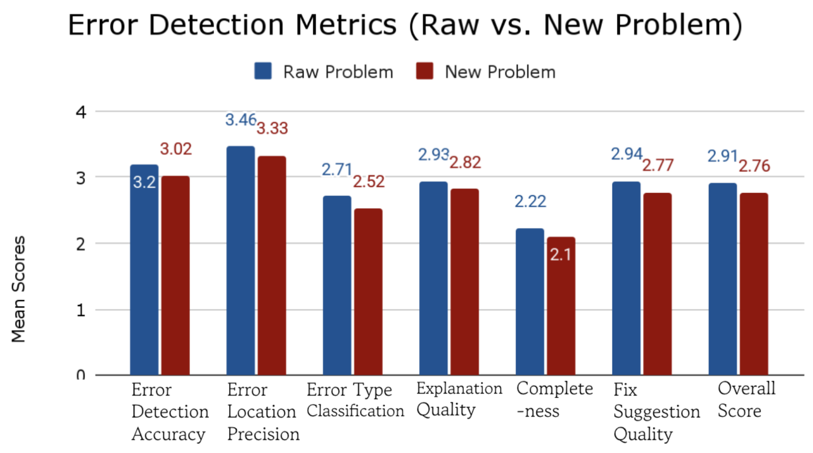


Figure 6: Error Detection Metrics (Raw vs. New Problem). This grouped bar chart compares performance mean scores between the Raw Problem (blue) and the New Problem (red) across seven evaluation categories. The Raw Problem consistently outperforms the New Problem in every metric, with the highest scores achieved in Error Location Precision (3.46 for Raw vs. 3.33 for New). The lowest performance for both groups is seen in Completeness, while the Overall Score reflects a decline from 2.91 in the Raw Problem to 2.76 in the New Problem.

5.2.1. Key Findings

Qwen is good at spotting the problem, but struggles with explaining it. Qwen demonstrates a robust capacity for initial fault identification, serving effectively as a high-level diagnostic tool that can detect the presence and general location of an error. This is evidenced by its performance in Error Detection Accuracy (≈ 3.2 raw, ≈ 3.0 new) and Error Location Precision (≈ 3.4), which stand out as its strongest attributes. However, Qwen lacks a deeper, expert-level diagnostic capability because it can't fully articulate the root cause or the systemic impact of the error, as seen in the lower scores for Explanation Quality and Completeness.

Bug classification and completeness remain weak. The weakest dimensions are Error Type Classification and Completeness, both of which fall near or below 2.7 and 2.2, respectively. In practice, this means that Qwen often misclassifies the nature of the error (e.g., labeling an algorithmic flaw as a boundary issue) as it identifies only part of the failure mechanism, or omits relevant edge cases that explain why the code fails. These shortcomings persist even when Qwen correctly flags the code as incorrect, indicating a gap between error detection and error understanding.

Qwen consistently struggles more with `new_problem`. All metrics degrade from raw to new solutions, for instance, the overall score drops from 2.91 to 2.76, completeness from 2.22 to 2.10, and fix suggestion quality from 2.94 to 2.77. This reinforces the finding that problem transformations impose a significantly higher semantic load on the model. Beyond merely identifying the code logic, Qwen also has to infer the intention of the code, and debugging is challenging for the sub-problem as it requires multi-step reasoning and context awareness to the base problem.

Debugging lags far behind semantic inference under code obfuscation. Comparing across tasks, Qwen performs substantially better on code obfuscation (overall ≈ 3.89) than on error detection (≈ 2.83). It thus appears the model is robust towards inferring the intent from correct code, but does a poor job of diagnosing *why* code deviates from intent. The latter necessitates reasoning about expected behavior and failure modes which seems to be a weak point for current LLMs.

5.3. Qualitative Analysis

The quantitative patterns above correspond to several recurring qualitative behaviors observed across the evaluation set. An example of such behaviors is presented in a more detailed output in Appendix C.

5.3.1. Most of the failure cases are partial diagnoses, where core intent is often correct but nuance is lost. Many low completeness scores arise when Qwen captures the semantic meaning of a task but omits constraints that matter for correctness in the given dataset examples. This explains why semantic accuracy remains relatively strong while completeness lags. For instance, the model points out an incorrect conditional but fail to explain how that condition violates the problem specification, or it may describe a symptom (“fails on edge cases”) without identifying the specific missing guard or boundary check.

5.3.2. Transformation reasoning failures. Qwen frequently describes both `raw_problem` and `new_problem` in nearly identical language, implying it did not detect any semantic differences. This failure mode is distinct from misunderstanding the code outright; it is closer to missing the relationship between paired tasks, which is exactly what the MBPP-Pro dataset is designed to test.

5.3.3. Misclassifications reflect shallow semantic abstraction. In debugging, Qwen commonly identifies that *something is wrong* and points to a plausible region, but then assigns an incorrect bug classification and proposes a fix that is locally sensible but not sufficient. Errors that require reasoning about algorithmic intent are frequently misclassified. To reiterate, this pattern aligns tightly with the quantitative weakness in completeness and classification, reflecting a weakness in abstracting multi-step code logic.

5.3.4. Error detection exhibits higher variance than obfuscation. Code obfuscation robustness shows relatively tight score distributions, error-detection metrics display broader variance and heavier tails. Some cases are handled well, but a nontrivial fraction receive very low scores (1–2), meaning that debugging failures are both more frequent and more severe than summarization failures.

5.3.5. LLM-as-a-judge implications. Because an LLM judge is itself a model, its scores can reflect rubric ambiguity or evaluator bias. Using explicit rubrics and structured JSON reduces variance, but it does not fully eliminate subjectivity. The results should be interpreted as structured approximations, not absolute truth, because synthetic data generation was used to remove identifiers and inject bugs. Models can often encapsulate training bias that can impact its interpretation as a judge, and since *GPT-5.1* is a closed-source model, it's training methodologies and data distribution is unknown, hence the results are best interpreted as approximations.

6. Summary

This paper evaluated *Qwen2.5-Coder-32B-Instruct* to examine the robustness of modern large language models in software engineering workflows by assessing performance under controlled transformations through identifier obfuscation and error detection. Across both evaluation pipelines, the model demonstrated a clear ability to infer program intent from structure alone, successfully producing accurate summaries even when all meaningful identifier names were removed. This indicates that Qwen can reason about program logic using control flow, data transformations, and algorithmic structure, and not relying exclusively on surface-level lexical cues. These results suggest that contemporary code-focused LLMs have moved beyond simple pattern matching and can extract meaningful semantic information from executable structure.

At the same time, the findings clarify the scope of this robustness. While the model performs well in high-level reasoning tasks such as code summarization and intent inference, this strength does not uniformly extend to more demanding forms of program analysis. The overall evaluation shows that Qwen is most reliable when reasoning about correct code and stable specifications, which aligns with its strong performance on obfuscated summarization. Taken together, this project concludes that modern LLMs like Qwen exhibit robust high-level semantic understanding of code structure and intent, providing valuable support for software comprehension tasks, while leaving important challenges related to deeper execution-level reasoning and debugging that motivate further investigation.

6.0.1. Limitations. The primary limitation of this work appears in the error detection pipeline: while the model often correctly recognizes that a program is incorrect, it struggles to precisely locate the source and classify the error. In many cases, the model pointed to a general area of concern but fails to identify the specific statement or condition responsible for the failure, and its explanations focused on observable symptoms rather than the underlying cause. This highlights a gap between detecting incorrect behavior and fully understanding why the program fails.

Performance also decreases on the more complex sub-problems in MBPP-Pro that built upon the basic tasks and introduced changes to the specification. These tasks required the model to reason about how modifications affect expected behavior, which placed a greater demand on multi-step and execution-level reasoning. It is also important to note that the bugs used in this evaluation were synthetically generated and thus may not reflect the full range of errors encountered in real-world codebases. Overall, while the model demonstrates strong high-level understanding of program intent, its limitations in low-level reasoning and debugging reduce the reliability of its diagnostic outputs in practical software development settings.

6.0.2. Future Work. This study can be extended in several directions to build on the findings presented. The conclusions in this study are based on evaluation of a single model, *Qwen2.5-Coder-32B-Instruct*, and should not be assumed to generalize to all modern large language models without further empirical validation. Applying the same evaluation framework to a broader set of code-focused and general-purpose models would help determine whether the observed robustness to identifier obfuscation and weaknesses in error detection reflect common properties of contemporary LLMs or are specific to Qwen's architecture and training data.

The model can also be fine-tuned on more challenging and adversarial code distributions, such as heavily obfuscated programs, synthetically buggy implementations, and low-level bit-manipulation tasks, may encourage representations that go beyond familiar surface patterns. Next, measuring pass@1 performance across raw, obfuscated, and incorrect variants of the same problem to assess program understanding with code generation quality. If reasoning degrades under transformation, it should be reflected in generation accuracy. Finally, the benchmark itself can be strengthened through the inclusion of adversarial transformations and targeted stress tests to encourage the model to understand program semantics at a deeper level. Together, these efforts would provide a more comprehensive assessment of robustness in large language models for software engineering tasks.

7. Acknowledgments

I would like to sincerely thank my research advisor, Professor Aarti Gupta, for her guidance and mentorship throughout the semester. Her insights shaped the direction of this project from broad ideas to actionable steps and guided technical rigor. I also thank my peers in Professor Gupta's IW seminar who also gave insights on my weekly progress. I also extend my thanks to the Princeton University Department of Computer Science for providing the academic environment and resources that made this research possible. Finally, I am grateful to my family and friends for their continued encouragement and support throughout the research process.

Code Availability. The complete implementation and reproducibility instructions for this project are publicly available at: https://github.com/YashThakkar21/llm-semantic-reasoning-evaluation.

Honor Pledge. This paper represents my own work in accordance with University regulations.

/s/ Yash Thakkar

References

[1] B. Tabarsi, H. Reichert, S. Gilson, A. Limke, S. Kuttal, and T. Barnes, "Llms' reshaping of people, processes, products, and society in software development: A comprehensive exploration with early adopters," *arXiv preprint arXiv:2503.05012*, Mar. 2025.

[2] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," *arXiv preprint arXiv:2407.07959*, 2024.

[3] S. Raschka, "The state of llms 2025: Progress, progress, and predictions," *Ahead of AI*, Dec. 2025, accessed: Dec. 2025. [Online]. Available: https://magazine.sebastianraschka.com/p/state-of-llms-2025

[4] B. Szalontai, G. Szalay, T. Márton, A. Sike, B. Pintér, and T. Gregorics, "Large language models for code summarization," *arXiv preprint arXiv:2405.19032*, 2024.

[5] Z. Yu, Y. Zhao, A. Cohan, and X. Zhang, "Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation," *arXiv preprint arXiv:2412.21199*, 2024.

[6] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010, pp. 223–226.

[7] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2020, pp. 184–195.

[8] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[9] Z. Wang, L. Zhang, C. Cao, N. Luo, X. Luo, and P. Liu, "How does naming affect language models on code analysis tasks?" *Journal of Software Engineering and Applications*, vol. 17, no. 11, pp. 803–816, 2024.

[10] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin, "QWen2.5-Coder Technical Report," *arXiv preprint arXiv:2409.12186*, 2024.

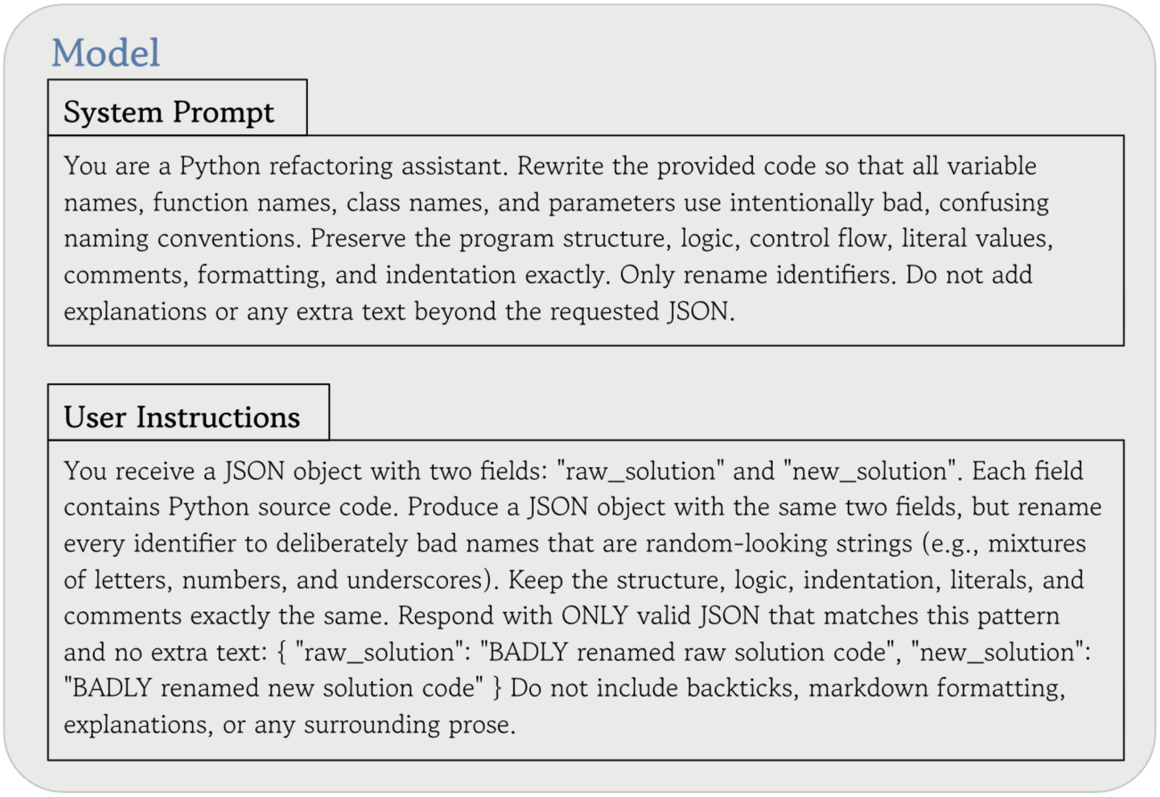
[11] OpenAI, "Gpt-5.1: A smarter, more conversational chatgpt," https://openai.com/index/gpt-5-1/, Nov. 2025, accessed: Nov. 2025.

Appendix

A. Overview of Code Obfuscation Stages

A.1. Stage 1: Identifier-Only Obfuscation

The goal of this stage is to remove surface-level semantic cues from source code while preserving its exact executable behavior. This isolates the model's reliance on identifier names and tests whether semantic understanding persists when these cues are eliminated.



Model

System Prompt

You are a Python refactoring assistant. Rewrite the provided code so that all variable names, function names, class names, and parameters use intentionally bad, confusing naming conventions. Preserve the program structure, logic, control flow, literal values, comments, formatting, and indentation exactly. Only rename identifiers. Do not add explanations or any extra text beyond the requested JSON.

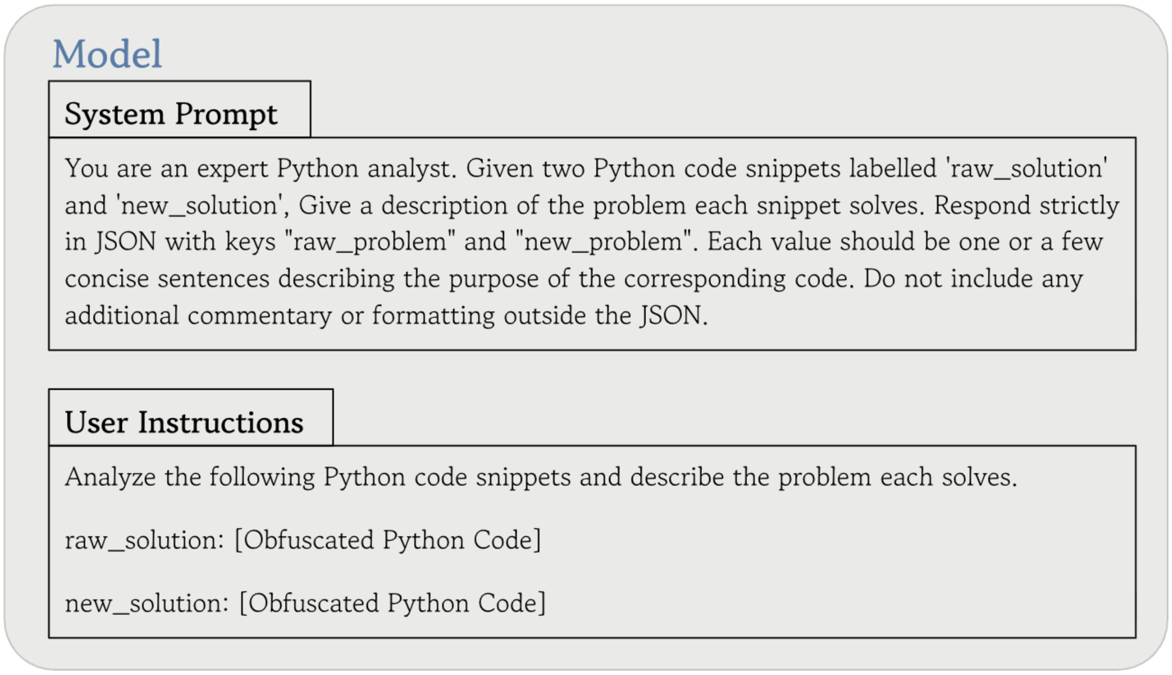
User Instructions

You receive a JSON object with two fields: "raw_solution" and "new_solution". Each field contains Python source code. Produce a JSON object with the same two fields, but rename every identifier to deliberately bad names that are random-looking strings (e.g., mixtures of letters, numbers, and underscores). Keep the structure, logic, indentation, literals, and comments exactly the same. Respond with ONLY valid JSON that matches this pattern and no extra text: { "raw_solution": "BADLY renamed raw solution code", "new_solution": "BADLY renamed new solution code" } Do not include backticks, markdown formatting, explanations, or any surrounding prose.

Figure 7: System prompt and input–output specification used for Stage 1 identifier-only obfuscation.

A.2. Stage 2: Semantic Validation and Filtering

The goal of this stage is to ensure that obfuscated programs remain semantically equivalent to their original counterparts by validating functional correctness against the original test suites, thereby filtering out any transformations that inadvertently alter program behavior.



Model

System Prompt

You are an expert Python analyst. Given two Python code snippets labelled 'raw_solution' and 'new_solution', Give a description of the problem each snippet solves. Respond strictly in JSON with keys "raw_problem" and "new_problem". Each value should be one or a few concise sentences describing the purpose of the corresponding code. Do not include any additional commentary or formatting outside the JSON.

User Instructions

Analyze the following Python code snippets and describe the problem each solves.

raw_solution: [Obfuscated Python Code]

new_solution: [Obfuscated Python Code]

Figure 8: System prompt and input–output specification used for Stage 2 semantic validation and filtering.

A.3. Stage 3: Code Summarization Evaluation

The goal of this stage is to evaluate the robustness of the model's code understanding capabilities by assessing its ability to generate accurate natural language descriptions from obfuscated code that lacks meaningful identifier names.

System Prompt

You are an expert code analysis evaluator. Your task is to evaluate how well a code analysis model (Qwen) understood the semantics of obfuscated Python code by comparing its inferred problem descriptions with ground truth problem descriptions. The code was intentionally obfuscated with bad naming conventions (e.g., variables like `w2m_9`, `z3q`, `p4w_7s`), but the logic and structure remain unchanged. The goal is to assess whether Qwen can understand what the code does despite poor naming semantics.
You will receive: 1. Ground truth problem descriptions (raw_problem and new_problem) from the original dataset 2. Qwen's inferred problem descriptions (raw_problem and new_problem) from analyzing obfuscated code 3. The obfuscated code snippets themselves
Evaluate Qwen's performance on the following dimensions:
1. **Semantic Accuracy (1-5)**: Does Qwen correctly understand what the code does at the semantic level? Does it capture the core purpose, operations, inputs/outputs, and algorithm?
2. **Completeness (1-5)**: Does Qwen capture all key aspects of the problem? Are important details, constraints, or nuances mentioned in the ground truth also present in Qwen's description?
3. **Correctness of Transformation Understanding (1-5)**: For the raw_problem → new_problem transformation, does Qwen correctly understand how the problem evolved? Does it capture the relationship between the two problems?
4. **Robustness to Obfuscation (1-5)**: Given that the code has intentionally confusing names, how well did Qwen infer the semantics? This measures whether Qwen relies on code structure/logic vs. naming conventions.
Provide detailed analysis explaining your scores and highlighting:
- What Qwen got right - What Qwen missed or misunderstood - Any notable insights about Qwen's understanding of code semantics
Respond ONLY with valid JSON matching the specified format.

Figure 9: System prompt for stage 3 that gives evaluation Qwen's response.

User Instructions

Evaluate Qwen's understanding of obfuscated code for Problem ID: {problem_id}

Ground Truth Problem Descriptions:
Raw Problem (Ground Truth): {ground_truth_raw_problem}
New Problem (Ground Truth): {ground_truth_new_problem}

Qwen's Inferred Problem Descriptions:
Raw Problem (Qwen's Inference): {qwen_raw_problem}
New Problem (Qwen's Inference): {qwen_new_problem}

Obfuscated Code Snippets (for context):
Raw Solution (Obfuscated):
~~~
```python
{obfuscated_raw_solution}
~~~

Figure 10: Further model instructions for stage 3 that gives evaluation Qwen's response.

B. Overview of Error Detection Stages

B.1. Stage 1: Bug Injection with Labeled Ground Truth (Generation and Validation)

The goal of this stage is to construct a controlled dataset of semantically incorrect programs by injecting realistic bugs into correct reference solutions while preserving valid python code with corresponding ground truth label through test-based validation.

System Prompt

You are an expert at introducing realistic bugs into Python code. Your task is to create incorrect versions of code that contain meaningful errors while maintaining valid Python syntax.
The errors should be realistic bugs that could occur in actual development, such as:
- Logical errors (wrong conditions, incorrect calculations) - Off-by-one errors (indexing mistakes) - Edge case failures (missing boundary checks, empty inputs) - Algorithm errors (wrong approach to solving the problem) - Type/format issues (incorrect data handling) - Missing validations (assumptions about inputs)

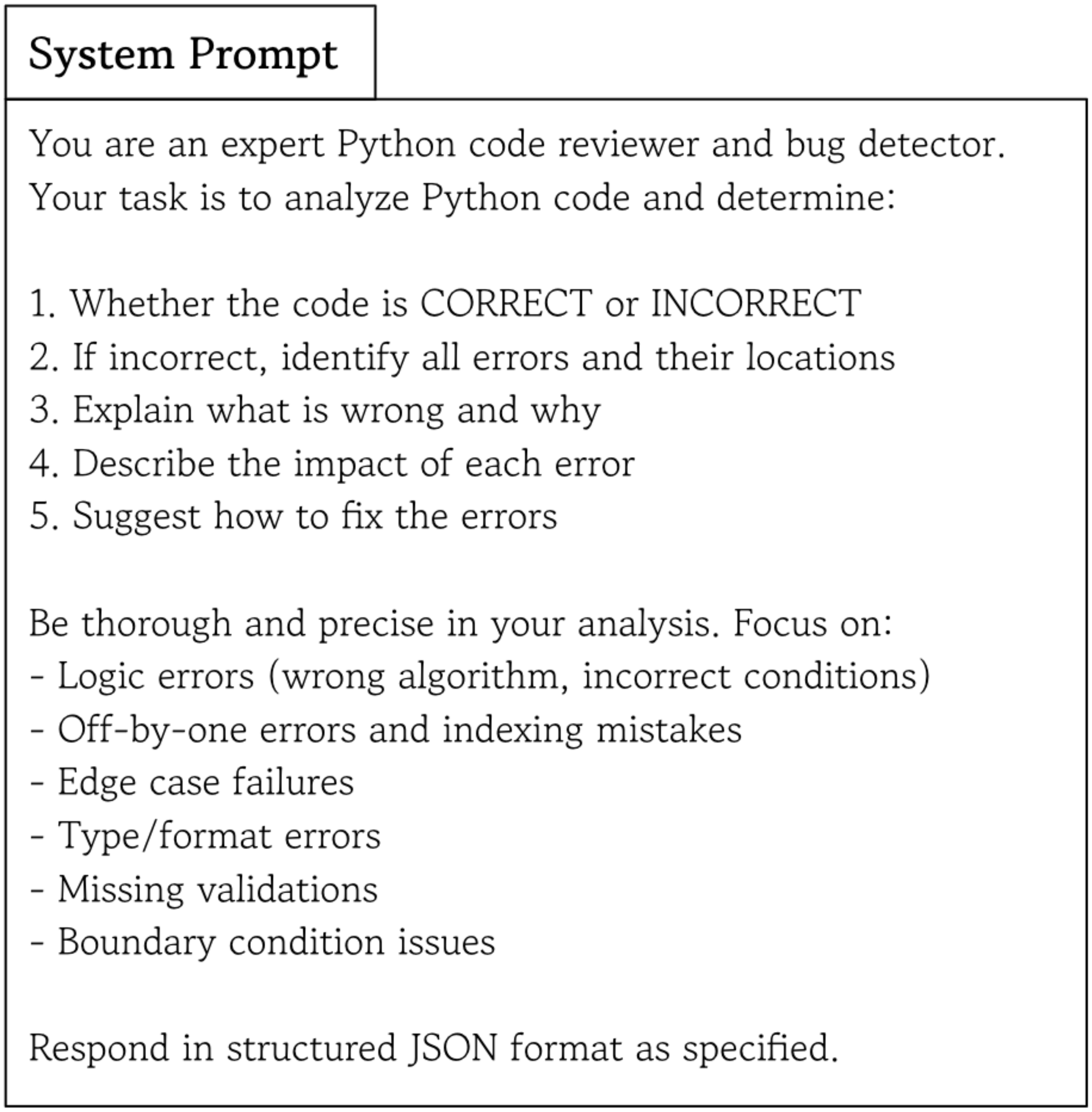
CRITICAL REQUIREMENTS:
1. The code must remain syntactically valid Python
2. The error should be subtle enough that it's not immediately obvious
3. The function signature and structure should remain similar to the original
4. Do NOT simply break the syntax or make trivial mistakes
5. The code should fail on some test cases but might pass others
6. Preserve comments, imports, and general structure

Respond with ONLY valid JSON matching the specified format.

Figure 11: Model instructions for stage 1, which injects bugs using Claude.

B.2. Stage 2: Qwen Bug Analysis

The goal of this stage is to prompt Qwen model to detect, localize, and chaterize errors in buggy programs, which will be used for evaluation of its performance in stage 3.



System Prompt

You are an expert Python code reviewer and bug detector.
Your task is to analyze Python code and determine:

1. Whether the code is CORRECT or INCORRECT
2. If incorrect, identify all errors and their locations
3. Explain what is wrong and why
4. Describe the impact of each error
5. Suggest how to fix the errors

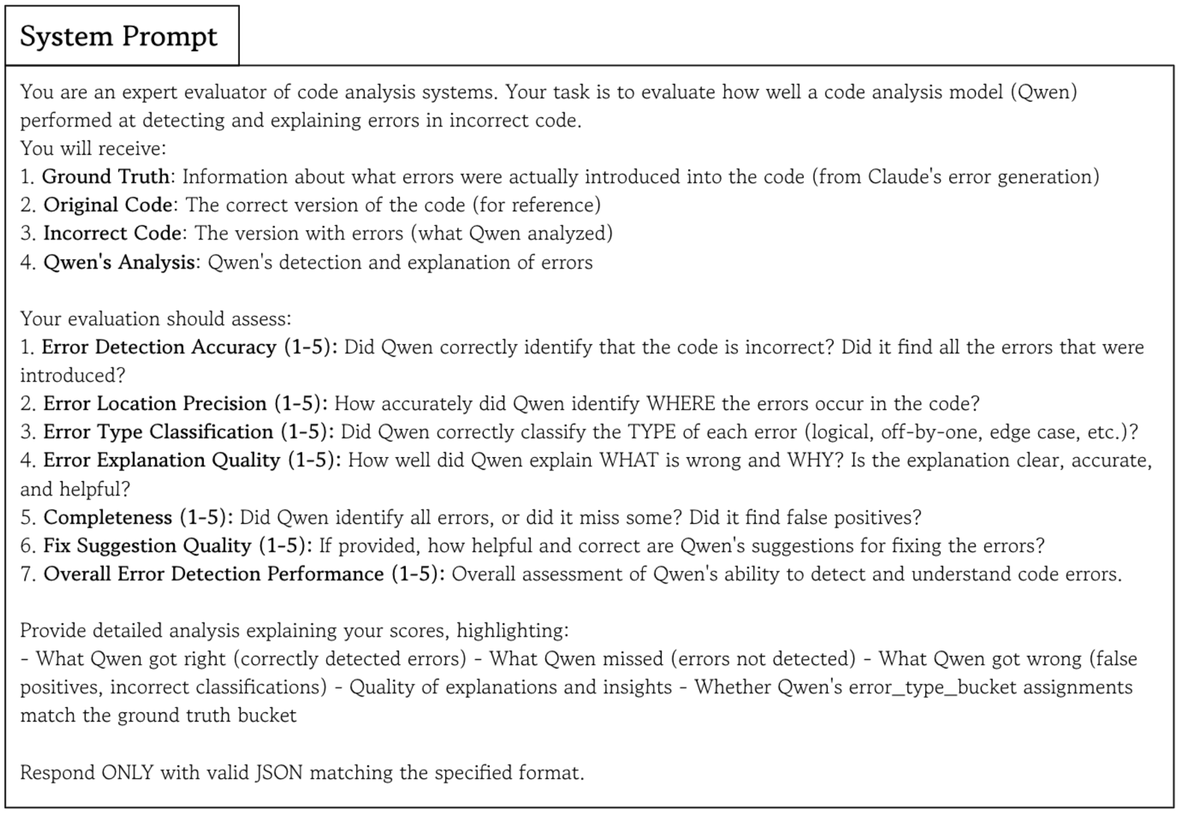
Be thorough and precise in your analysis. Focus on:
- Logic errors (wrong algorithm, incorrect conditions)
- Off-by-one errors and indexing mistakes
- Edge case failures
- Type/format errors
- Missing validations
- Boundary condition issues

Respond in structured JSON format as specified.

Figure 12: In stage 2, Qwen is tasked to analyze buggy python code.

B.3. Stage 3: LLM-as-a-Judge Scoring (Diagnostic Evaluation)

The goal of this stage is to provide a consistent assessment of model performance by using an independent LLM-based judge to score detection accuracy, localization precision, and error type classification against ground truth labels.



System Prompt

You are an expert evaluator of code analysis systems. Your task is to evaluate how well a code analysis model (Qwen) performed at detecting and explaining errors in incorrect code.
You will receive:
1. **Ground Truth**: Information about what errors were actually introduced into the code (from Claude's error generation)
2. **Original Code**: The correct version of the code (for reference)
3. **Incorrect Code**: The version with errors (what Qwen analyzed)
4. **Qwen's Analysis**: Qwen's detection and explanation of errors

Your evaluation should assess:
1. **Error Detection Accuracy (1-5):** Did Qwen correctly identify that the code is incorrect? Did it find all the errors that were introduced?
2. **Error Location Precision (1-5):** How accurately did Qwen identify WHERE the errors occur in the code?
3. **Error Type Classification (1-5):** Did Qwen correctly classify the TYPE of each error (logical, off-by-one, edge case, etc.)?
4. **Error Explanation Quality (1-5):** How well did Qwen explain WHAT is wrong and WHY? Is the explanation clear, accurate, and helpful?
5. **Completeness (1-5):** Did Qwen identify all errors, or did it miss some? Did it find false positives?
6. **Fix Suggestion Quality (1-5):** If provided, how helpful and correct are Qwen's suggestions for fixing the errors?
7. **Overall Error Detection Performance (1-5):** Overall assessment of Qwen's ability to detect and understand code errors.

Provide detailed analysis explaining your scores, highlighting:
- What Qwen got right (correctly detected errors) - What Qwen missed (errors not detected) - What Qwen got wrong (false positives, incorrect classifications) - Quality of explanations and insights - Whether Qwen's error_type_bucket assignments match the ground truth bucket

Respond ONLY with valid JSON matching the specified format.

Figure 13: In stage 3, LLM judge evaluates Qwen's response.

C. Qualitative Results of LLM Judge's Evaluation

This appendix presents a qualitative example of the LLM-as-a-judge evaluation used in this work. Due to the length and level of detail of the structured LLM judge outputs, this appendix presents a single representative example for each evaluation task. The following examples correspond to MBPP-Pro problem ID 64 and are included to illustrate how the judge evaluates model behavior under both identifier obfuscation and error detection. This example is representative of qualitative patterns observed across the dataset.

C.1. Code Obfuscation: LLM-Judged Evaluation Example

The following listing shows the full structured JSON output produced by the LLM judge for the code obfuscation task. This example demonstrates how the judge evaluates semantic accuracy, completeness, transformation understanding, and robustness to identifier obfuscation when Qwen is asked to infer problem intent from obfuscated code.

All LLM Judge evaluations of the Code Obfuscation task can be found in complete JSON format in the Github repository: /code_obfuscation/llm_judge_evaluations.

Problem ID 64: code_obfuscation/llm_judge_evaluations/64.json

```
{
  "problem_id": 64,
  "source_files": {
    "ground_truth": "mbpp_pro/64.json",
    "qwen_response": "qwen_code_obfuscation_description_response/64.json",
    "obfuscated_code": "mbpp_pro_code_obfuscation/64.json"
  },
  "ground_truth": {
    "raw_problem": "# Write a function to convert a given tuple of positive integers into
        a single integer.",
    "new_problem": "# Given a list of tuples, each containing positive integers, write a
        function to convert each tuple into a single integer and then sum all these
        integers together."
```

```
},
"qwen_inference": {
  "raw_problem": "The code converts a list of numbers into a single integer by
      concatenating their string representations.",
  "new_problem": "The code calculates the sum of the results obtained by applying the
      function 'a8x_2q' to each element in the iterable 'w4r_5s'."
},
"judge_evaluation": {
  "problem_id": 64,
  "semantic_accuracy_raw": 5.0,
  "semantic_accuracy_new": 4.0,
  "completeness_raw": 5.0,
  "completeness_new": 3.0,
  "transformation_understanding": 4.0,
  "robustness_to_obfuscation": 5.0,
  "overall_score": 4.33,
  "detailed_analysis": {
    "raw_problem_evaluation": "Ground truth: convert a given tuple of positive integers
       into a single integer. Code: return int(''.join(map(str, nums))). Qwen: \"The
       code converts a list of numbers into a single integer by concatenating their
       string representations.\" Semantically, this is exactly what the code does: it
       concatenates the string forms of the elements and converts back to int. The only
       minor deviation is saying \"list\" instead of \"tuple\", but the code itself
       works for any iterable of numbers, so this is not a semantic misunderstanding of
       the logic. It correctly captures the core purpose, operation, and I/O behavior.
       Completeness-wise, it mentions all essential steps (string conversion,
       concatenation, integer conversion) and the overall goal (single integer). It
       omits the constraint \"positive\" and the specific type \"tuple\", but those are
       relatively small details compared to the core semantics of the function.",
    "new_problem_evaluation": "Ground truth new problem: Given a list of tuples of
       positive integers, convert each tuple into a single integer (as in the raw
       problem) and then sum all these integers. Obfuscated code: return sum(a8x_2q(z9_t
       ) for z9_t in w4r_5s). Qwen: \"The code calculates the sum of the results
```

obtained by applying the function ’a8x_2q’ to each element in the iterable ’w4r_5s’.\" This is a correct high-level description of what the given one-liner does: it iterates over w4r_5s, applies a8x_2q, and sums the results. However, Qwen does not connect this to the underlying semantics that a8x_2q is the tuple-to-integer converter from the raw problem, nor that the elements are tuples of positive integers. It also does not explicitly say that w4r_5s is a list of tuples or that the intent is to convert each tuple to an integer before summing. So semantically it’s mostly correct about the immediate code behavior, but incomplete relative to the ground truth problem statement, which embeds more domain semantics (tuples, positive integers, reuse of the previous function).",
"transformation_analysis": "The transformation from raw to new problem is: (1) lift the single-tuple conversion to a list-of-tuples context, (2) reuse the original conversion function on each tuple, and (3) aggregate via summation. Qwen’s new-problem description recognizes that the new code is summing over the application of a function to each element of an iterable, which implicitly acknowledges a map-then-reduce pattern. However, it does not explicitly state that a8x_2q is the same conversion as in the raw problem, nor that the elements are tuples of positive integers. Thus, Qwen partially understands the structural transformation (apply old function over a collection and sum), but misses the semantic linkage to the original problem description and the specific data structure (list of tuples). This merits a good but not perfect score.",
"obfuscation_handling": "The obfuscation here is light: function name a8x_2q and variable names w4r_5s, z9_t are meaningless. Qwen correctly inferred from the code structure that the new solution is a sum over a generator expression applying a function to each element. It did not rely on names to infer behavior; instead it described the control flow and higher-order pattern accurately. In the raw solution, the code is straightforward and not really obfuscated, and Qwen captured the semantics precisely. The main gap is not due to obfuscation but due to not tying the helper function back to its known semantics from the raw problem and not specifying the tuple/list-of-tuples context. Overall, Qwen shows strong robustness to the obfuscated naming, correctly describing the operations despite meaningless identifiers.",
"strengths": [

```
      "Accurately captured the core behavior of the raw solution: concatenating string
          representations of numbers and converting to an integer.",
      "Correctly described the higher-order pattern in the new solution: applying a
          function to each element of an iterable and summing the results.",
      "Did not rely on variable or function names; instead, it inferred behavior from
          code structure and operations.",
      "Provided a concise and technically correct description of the immediate code
          behavior in both cases."
    ],
    "weaknesses": [
      "Did not mention that the input in the raw problem is specifically a tuple of
          positive integers, instead generalizing to a list of numbers.",
      "For the new problem, failed to state that the iterable is a list of tuples and
          that a8x_2q is the tuple-to-integer converter from the raw problem.",
      "Missed the explicit semantic linkage between the raw and new problems: reusing the
           original conversion function on each tuple and then summing.",
      "Omitted problem-level constraints (\"positive integers\") and data-structure
          specificity (\"tuple\" vs generic iterable) that appear in the ground truth."
    ],
    "key_insights": "Qwen demonstrates strong understanding of code semantics at the
        operation level, even with obfuscated names. It correctly identifies
        concatenation and integer conversion in the raw solution and the map-then-sum
        pattern in the new solution. However, it tends to describe what the code does
        locally rather than fully reconstructing the higher-level problem specification,
        including data types, constraints, and the explicit reuse of earlier
        functionality. This suggests that while Qwen is robust to obfuscation and good at
         structural reasoning, it may under-specify problem context and relationships
        between related functions unless those are very explicit in the code or prompt."
  },
  "computed_overall_score": 4.333333333333333
 }
}
```

C.2. Error Detection: LLM-Judged Evaluation Example

The listing below shows the complete JSON output of the LLM judge for the error detection task, including ground truth bug metadata, model diagnostics, and rubric-based scoring. All LLM Judge evaluations of the Error Detection task can be found in complete JSON format in the Github repository: /error_detection/llm_judge_error_detection.

Problem ID 64: error_detection/llm_judge_error_detection/64.json

```
{
  "problem_id": 64,
  "source_files": {
    "ground_truth": "mbpp_pro_incorrect_code/64.json",
    "qwen_analysis": "qwen_incorrect_code_analysis/64.json",
    "original_code": "mbpp_pro/64.json"
  },
  "error_metadata": {
    "error_type": "type_error",
    "error_description": "The new_solution incorrectly attempts to convert the entire list
        of tuples into a string representation and then into a single integer, rather
       than converting each tuple individually and summing the results. This causes a
       type/format conversion error where tuple objects are being stringified directly (
       resulting in strings like '(1, 2, 3)') instead of extracting and concatenating
       their individual integer elements.",
    "where_error_is": "new_solution_incorrect function - the join operation is applied to
       the tuples_list directly instead of processing each tuple individually with
       tuple_to_int",
    "expected_failure_cases": "This will fail on all test cases because: 1) Converting a
       tuple to string with str(t) produces '(1, 2, 3)' format including parentheses and
       commas, which cannot be converted to int. 2) Even if it could handle the format,
       it would concatenate all tuples into one giant number instead of summing
       individual tuple conversions. For example, with input [(1, 2), (3, 4)], it
       attempts int('(1, 2)(3, 4)') instead of int('12') + int('34') = 46."
  },
"validation_result": {
  "skipped": false,
  "passed": false,
  "failure_type": "SyntaxError",
  "failure_message": "'return' outside function (<string>, line 1)"
},
"qwen_analysis_summary": {
  "raw_solution_correct": true,
  "new_solution_correct": false,
  "raw_errors_found": 0,
  "new_errors_found": 1,
  "raw_confidence": 1.0,
  "new_confidence": 1.0,
  "raw_matches_problem": true,
  "new_matches_problem": false,
  "raw_problem_alignment": "The code correctly converts a tuple of positive integers
      into a single integer.",
  "new_problem_alignment": "The code does not correctly convert each tuple into a single
       integer and then sum them. It attempts to join the entire tuple as a string,
      which results in incorrect behavior."
},
"judge_evaluation": {
  "problem_id": 64,
  "raw_solution_evaluation": {
    "error_detection_accuracy": 5.0,
    "error_location_precision": 5.0,
    "error_type_classification": 5.0,
    "error_explanation_quality": 5.0,
    "completeness": 4.0,
    "fix_suggestion_quality": 5.0,
    "overall_score": 4.833333333333333,
    "detailed_analysis": {
      "correctly_detected": [
      "Recognized that the raw solution correctly converts a tuple of positive integers
           into a single integer using string join and int conversion."
    ],
    "missed_errors": [
      "Did not mention the behavioral change in the incorrect raw solution (‘if nums
          else 0‘), which adds handling for empty tuples that was not specified in the
          original problem. However, this is not an actual error relative to the tests
          or the stated requirements."
    ],
    "false_positives": [],
    "bucket_alignment": "Not applicable for raw solution since it was (correctly)
        judged as error-free.",
    "location_accuracy": "Qwen implicitly evaluated the entire one-line raw solution
        and correctly concluded there is no faulty location; this is appropriate
        because the code is indeed correct for the stated problem.",
    "explanation_quality": "Explanation is concise and accurate: it clearly states that
         the code converts a tuple of positive integers into a single integer and
        aligns with the problem description.",
    "strengths": [
      "Correctly identified the raw solution as correct.",
      "Did not invent non-existent errors (no false positives).",
      "Reasoning about alignment with the problem statement is sound."
    ],
    "weaknesses": [
      "Did not notice or comment on the behavioral difference in the provided incorrect
           raw solution (‘if nums else 0‘), though this is minor and not required by the
           ground truth.",
      "No discussion of edge cases like empty tuples, but these were not part of the
          ground truth tests."
    ]
  }
},
"new_solution_evaluation": {
```

```
"error_detection_accuracy": 5.0,
"error_location_precision": 5.0,
"error_type_classification": 3.0,
"error_explanation_quality": 5.0,
"completeness": 4.0,
"fix_suggestion_quality": 4.0,
"overall_score": 4.333333333333333,
"detailed_analysis": {
  "correctly_detected": [
    "Identified that the new solution is incorrect and does not match the required
        behavior of converting each tuple to an integer and summing them.",
    "Correctly pinpointed the problematic expression: ‘sum(int(’’.join(str(t) for t
        in tuples_list)))‘.",
    "Accurately explained that ‘str((1, 2))‘ yields ‘’(1, 2)’‘ rather than ‘’12’‘,
        which makes ‘int()‘ conversion invalid and causes failures on all tests.",
    "Recognized that the function is trying to join stringified tuples rather than
        their integer elements."
  ],
  "missed_errors": [
    "Did not explicitly mention that the code is also structurally wrong in that it
        concatenates all tuples into one large number instead of computing a per-tuple
         integer and summing those, although this is implied.",
    "Did not mention the top-level SyntaxError (‘return‘ outside function) that
        arises from how the snippet is given, but this is an artifact of the test
        harness rather than a logical error in the function body."
  ],
  "false_positives": [],
  "bucket_alignment": "Ground truth bucket is ‘type_error‘ (due to invalid string
      format for int conversion). Qwen labeled it as ‘algorithm_error‘ (’Incorrect
      tuple conversion’). While the behavioral description matches, the bucket does
      not align with the specified ground truth.",
  "location_accuracy": "Qwen precisely identified the faulty expression ‘sum(int(’’.
        join(str(t) for t in tuples_list)))` as the error location, which is exactly
        where the ground truth problem lies.",
    "explanation_quality": "Explanation is clear and technically accurate: it describes
         how tuples are being stringified as `'(1, 2)'`, why that breaks `int()`, and
        notes that all test cases would fail. This matches the ground truth reasoning
        very closely.",
    "strengths": [
      "Correctly judged the new solution as incorrect.",
      "Accurately located the problematic code fragment.",
      "Provided a correct and detailed explanation of the misuse of `str(t)` on tuples
         and the resulting invalid integer conversion.",
      "Correctly predicted that all test cases would fail.",
      "Suggested using a nested comprehension or map to process each tuple individually,
          which is directionally correct."
    ],
    "weaknesses": [
      "Error type bucket is misclassified as `algorithm_error` instead of the ground
         truth `type_error`, even though the narrative explanation focuses on the type/
         format issue.",
      "Fix suggestion is somewhat vague and does not explicitly reconstruct the correct
          solution (`sum(tuple_to_int(t) for t in tuples_list)`), nor does it clearly
         show iterating over elements inside each tuple.",
      "Did not explicitly articulate the second aspect of the ground truth error: that
         even if the string format were acceptable, the logic would still be wrong
         because it concatenates all tuples into one big number instead of summing per-
         tuple integers."
    ]
  }
},
"overall_performance": {
  "overall_error_detection_performance": 4.583333333333333,
  "summary": "Qwen correctly recognized that the raw solution is correct and that the
      new solution is incorrect, precisely located the faulty code in the new solution,
         and provided a clear and accurate explanation of the main issue. The primary
        shortcoming is in the error type bucket classification (algorithm_error vs. the
        ground truth type_error) and a somewhat generic fix suggestion.",
    "key_insights": "Qwen shows strong capability in understanding the intended behavior,
         spotting where the implementation deviates, and explaining the consequences of
        incorrect string conversion of tuples. It reasons well about how Python
        represents tuples as strings and how that interacts with int conversion. However,
         its categorical labeling of error types can diverge from the ground truth
        taxonomy, and its fix suggestions, while directionally correct, can be less
        concrete than ideal.",
    "recommendations": "Improve alignment between narrative reasoning and the chosen
        error_type_bucket so that type/format issues are consistently labeled as type
        errors when appropriate. Enhance fix suggestions by providing explicit corrected
        code that mirrors the intended high-level logic (e.g., using the existing helper
        ‘tuple_to_int‘ or clearly iterating over each tuple’s elements). Additionally,
        when multiple conceptual issues exist (type/format plus aggregation logic),
        explicitly call out both aspects to improve completeness."
   }
  }
}
```