Semester Long Internship Report

Autumn, 2025

On

Benchmarking Large Language Models for Logical Error Detection

Submitted by

Yuvraj Pandey

K. J. Somaiya School of Engineering, Mumbai

Under the guidance of

Dr. Kushal Shah

and

Prof. Prabhu Ramachandran

FOSSEE, Indian Institute of Technology Bombay

March 31, 2026

Declaration

I hereby declare that this internship report, entitled *"Benchmarking Large Language Models for Logical Error Detection"* is an original and authentic work submitted in partial fulfillment of the requirements for the FOSSEE Semester-Long Internship (Part-Time), conducted by the Free/Libre and Open Source Software for Education (FOSSEE) team at the Indian Institute of Technology Bombay.

All content herein is expressed in my own words, except where explicit references have been made to the ideas or works of others, which have been duly acknowledged and cited. No part of this submission has been plagiarised, misrepresented, fabricated, or falsified, nor has it been submitted to any other institution for the award of any degree or academic credit.

Name : Yuvraj Pandey

Internship : FOSSEE Semester-Long Internship, Autumn 2025

Guide : Dr. Kushal Shah

Date : March 31, 2026

Abstract

The ability of Large Language Models (LLMs) to identify logical errors in code is a promising avenue for automated feedback in programming education. This report presents the work carried out during a part-time semester internship with the FOSSEE team at IIT Bombay, focusing on a rigorous comparative benchmark of six LLMs, comprising both proprietary and open-weights models, on their ability to classify logical errors in student-written Python programs.

The study employs a ten-category Pedagogical-Granular (PG) error taxonomy covering error types such as Input, Output, Variable, Computation, Condition, Branching, Loop, Array/String, Function, and Conceptual errors. One hundred Python code snippets drawn from the Yaksh learning platform serve as the evaluation dataset. Models are evaluated under two paradigms: a *Single-Label* mode, where each model must identify the most dominant error, and a *Multi-Label* mode, where all applicable error types must be identified. Performance is measured using four complementary metrics, namely Precision (Case A), Recall (Case B), Any-Overlap (Case C), and the Jaccard Index (Case D), to provide a holistic view of model behaviour.

Key findings reveal that inter-model consensus degrades by roughly 50% when moving from the constrained single-label task to the open-ended multi-label task. GPT-5.2 emerges as the most reliable model with a Jaccard score of 57.1% in multi-label evaluation, while distinct *Conservative* (Gemini-2.5-flash) and *Aggressive* (Qwen3-Coder, DeepSeek-v3.2) behavioural profiles are identified. A near-universal bias toward identifying Conceptual errors (Category J) is documented across five of the six models. The results underscore that fully automated logical error detection at production-quality accuracy remains an open problem, and that human oversight is still essential for complex reasoning tasks in educational settings.

Acknowledgements

I would like to express my sincere gratitude to the entire FOSSEE team at IIT Bombay for providing me with the opportunity to participate in the Semester-Long Internship Programme. The welcoming and collaborative environment throughout the programme made this a truly enriching experience, and I am grateful to everyone who contributed to it.

I am deeply thankful to my mentors, **Dr. Kushal Shah** and **Prof. Prabhu Ramachandran**, for their guidance, thoughtful feedback, and continuous encouragement throughout the internship. Their insights were instrumental in framing the research questions and shaping the direction of this project.

I would also like to acknowledge **Ms. Usha Viswanathan**, **Ms. Vineeta Ghavri**, and **Mr. Prathamesh Salunke**, and other team members at FOSSEE, for their support during the programme. They were always approachable and helpful whenever questions or matters arose, and their assistance is sincerely appreciated.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

Programming education at scale presents a fundamental challenge: providing timely, precise feedback on logical errors to a large number of learners. Unlike syntax errors, which compilers and interpreters flag immediately, *logical errors* produce code that runs without crashing yet yields incorrect results. They are far harder to detect automatically because their presence depends on the *intent* of the program rather than its surface-level structure.

The advent of Large Language Models (LLMs) with strong code-understanding capabilities has opened a new avenue for automated pedagogical feedback. Models such as GPT, Claude, Gemini, and their open-weights counterparts have demonstrated impressive performance on a variety of code-related tasks. However, their behaviour on the nuanced task of *classifying* the type of logical error present in student code, as opposed to merely detecting that an error exists, is not yet well-characterised.

The FOSSEE (Free/Libre and Open Source Software for Education) project at IIT Bombay is engaged in developing open-source tools to support technical education in India. In this context, the present internship project was motivated by the following question: *can state-of-the-art LLMs reliably serve as automated logical error classifiers in a programming education pipeline?*

1.2 Problem Statement

Existing benchmarks for LLMs on code primarily focus on code generation, bug fixing, or binary correctness prediction. The specific task of **multi-class logical error classification** using a structured pedagogical taxonomy is less explored. Further, most evaluations test a single model in isolation; comparative studies across proprietary and open-weights models under identical conditions, with reproducible prompts and a standardised dataset, are rare.

This project addresses these gaps by:

1. Designing and running a reproducible inference pipeline across six leading LLMs.
2. Evaluating both constrained (single-label) and unconstrained (multi-label) classification modes.
3. Analysing inter-model agreement, behavioural profiles, and systematic biases.

1.3 Objectives

The specific objectives of this internship work were:

- To study and adapt an existing ten-category logical error taxonomy for use in LLM prompting.
- To build a robust, resume-safe benchmarking script that calls multiple LLMs through the OpenRouter API.
- To design effective single-label and multi-label prompt templates based on the taxonomy.
- To evaluate model outputs using four complementary metrics: Precision, Recall, Any-Overlap, and Jaccard.
- To analyse consensus, stability, behavioural profiles, and systematic biases across models.
- To document findings in a reusable analysis notebook and a final report.

1.4 Repository

The primary team repository, which integrates results across all three taxonomies explored by the full intern cohort, is publicly available at:

https://github.com/atmabodha/fossee-iitb

The work specific to this report, including the benchmarking script, prompts, dataset, and analysis notebook, is located at:

https://github.com/yxpx/fossee/tree/main/Yuvraj

2 Literature Review

2.1 Logical Errors in Programming Education

Logical errors have long been recognised as one of the primary obstacles for novice programmers [1]. Unlike syntactic or runtime errors, they require understanding of the programmer's intent, making them difficult to diagnose using traditional static analysis. Research in computing education has proposed various taxonomies for classifying such errors, primarily to guide instructional feedback.

Kohn proposed a hierarchical classification of Python beginner errors that distinguishes between semantic, conceptual, and structural mistakes [2]. More recently, structured taxonomies have been incorporated into intelligent tutoring systems to provide category-specific hints and corrective feedback.

2.2 LLMs for Code Understanding

The emergence of transformer-based models pre-trained on large code corpora, such as Codex, GPT-4, and DeepSeek-Coder, has significantly advanced automated code understanding. Studies have demonstrated that these models can generate syntactically correct code, perform bug detection in controlled settings, and explain code in natural language.

However, most evaluations measure *functional correctness* (i.e., whether generated code passes test cases) rather than *diagnostic accuracy* (i.e., whether the model can identify what kind of logical error is present). The distinction matters: a model might fix a bug without being able to articulate its category.

2.3 Prompt Engineering for Classification

Recent work on prompt engineering has shown that providing structured taxonomies and few-shot examples in the prompt significantly improves LLM classification accuracy. Chain-of-Thought (CoT) and Tree-of-Thought (ToT) prompting have both demonstrated gains in structured reasoning tasks.

The primary reference for the taxonomy and prompting strategy used in this project is Lee et al. [3], who found that embedding error descriptions directly in the prompt raised average classification accuracy by 21 percentage points over a baseline without such descriptions. That study also established the ten-category PG taxonomy and the priority ordering used to resolve classification ambiguity, both of which form the backbone of the present work.

2.4 Multi-Label Classification with LLMs

Multi-label classification, i.e., assigning a subset of labels from a fixed set, is intrinsically harder than single-label classification, as it requires a model to reason about the co-occurrence and independence of multiple categories. For logical errors in particular, a single code snippet may simultaneously exhibit a Computation error embedded within a Condition block, making boundary cases unavoidable. Prior work on this task is sparse, and standardised evaluation metrics such as the Jaccard Index are rarely applied to LLM outputs in this domain.

3 Error Taxonomy

3.1 Pedagogical-Granular (PG) Taxonomy

This project adopts the **Pedagogical-Granular (PG)** taxonomy, originally introduced by Lee et al. [3], to provide fine-grained, instructionally meaningful categories for logical errors in Python programs. The taxonomy defines ten error types, labelled A through J, plus a special `N` (No Error) label. Each category corresponds to a distinct aspect of program structure and is intended to be mutually exclusive at the level of primary cause.

Table 1: The ten-category PG error taxonomy used in this study.

ID	Category	Description
A	Input	Failing to receive all input values, or using the incorrect data type for variables.
B	Output	Non-compliance with required output formats, or outputting incorrect string literals.
C	Variable	Storing incorrect values, or specifying the wrong data type for a variable.
D	Computation	Calculating with incorrect values, or using the wrong arithmetic or logical operations.
E	Condition	Incorrect or insufficient conditional expressions in control structures.
F	Branching	Incorrect program flow (e.g., wrong break placement, or `if-if` instead of `if-else`).
G	Loop	Incorrect conditions or variables used in loop declarations.
H	Array/String	Incorrect initialisation of arrays/strings, or referencing an incorrect index.
I	Function	Incorrectly defined parameters or return values, or wrong arguments passed in calls.
J	Conceptual	Solving the wrong problem entirely, or missing necessary loops or conditions.
N	No Error	No logical error is present in the submitted code.

3.2 Higher-Level Groupings

The ten categories can be organised into five higher-level conceptual groups, which are useful for aggregate analysis:

Table 2: Higher-level category groupings for the PG taxonomy.

Group	Includes	Rationale
Problem Understanding and Algorithm Design	J	Errors from misunderstanding the problem or designing an incorrect solution approach.
Interface and Contract	A, B, I	Violations of input, output, or function interface specifications.
State Representation	C, H	Incorrect storage, initialisation, or referencing of program data.
Control Logic	E, F, G	Errors that disrupt execution flow via incorrect conditions, branches, or loops.
Computation and Expression	D	Incorrect calculations, operators, or expressions applied to otherwise valid data.

3.3 Priority Ordering and Error Relationships

A key challenge in logical error classification is that errors are not always independent: an upstream Input error can cascade into apparent Computation or Output errors. The taxonomy addresses this through two concepts:

- **Causal relationships**: An error at an early program stage (e.g., incorrect input handling) causes apparent errors at later stages.
- **Coincidence relationships**: Overlapping definitions, such as a computation error occurring within a condition declaration.

To resolve ambiguity in single-label scenarios, the following priority ordering is established [3]:

$$\mathrm{J} > \mathrm{I} > \mathrm{A} > \mathrm{C} > \mathrm{H} > \mathrm{E} = \mathrm{G} > \mathrm{D} > \mathrm{F} > \mathrm{B}$$

Conceptual errors (J) have the highest priority because they represent the broadest class of misunderstanding and overlap with all other categories.

4 Methodology

4.1 Dataset

The evaluation dataset, `yaksh100.json`, comprises **100 Python code snippets** drawn from the Yaksh learning platform. Each item contains a natural-language problem description and a student-submitted answer code. Human-annotated Ground Truth (GT) labels were produced for each snippet by a domain expert and stored in `results/csv/ground.csv`. GT labels were assigned using the PG taxonomy; multi-label ground truth is provided (e.g., a snippet may be labelled `J,I,B`), reflecting the reality that a single program can contain multiple co-occurring logical errors. Where a snippet contains no logical error, it is labelled `N`.

4.2 Models Evaluated

Six LLMs were evaluated, representing a mix of proprietary (closed-source) and open-weights architectures:

Table 3: LLMs evaluated in the benchmark.

Model	OpenRouter Identifier	Provider	Type
GPT-5.2	`openai/gpt-5.2`	OpenAI	Closed
Claude Sonnet 4.5	`anthropic/claude-4.5-sonnet`	Anthropic	Closed
Gemini 2.5 Flash	`google/gemini-2.5-flash`	Google	Closed
DeepSeek-v3.2	`deepseek/deepseek-v3.2`	DeepSeek	Open
Qwen3-Coder	`qwen/qwen3-coder`	Alibaba	Open
GPT-OSS-120b	`openai/gpt-oss-120b`	OpenAI	Open

All models were accessed via the **OpenRouter** API, which provides a unified interface to multiple model providers. Temperature was set to 0 for all models to ensure deterministic outputs. Reasoning mode was disabled for all models except Qwen3-Coder.

4.3 Classification Modes

Each model was evaluated under two distinct prompt paradigms:

Single-Label Mode

The model must identify the *single most dominant* logical error. The output is constrained to exactly one label from {A, B, C, D, E, F, G, H, I, J, N}. This corresponds to a standard multi-class classification task.

Multi-Label Mode

The model must identify *all applicable* error types. The output is a comma-separated list (e.g., `A,D,J`). If no error is present, the model should output `N` alone. This open-ended task tests the model's ability to reason about multiple co-occurring errors simultaneously.

Each mode was run twice on the full dataset (Run 1 and Run 2) to measure run-to-run stability.

4.4 Prompt Design

The prompts were designed to be structured, minimal, and taxonomy-driven. Both prompts include: a role declaration, a task description, the full PG taxonomy in XML format with descriptions for all ten categories, a reasoning instruction block, strict output format rules, and the student code snippet injected via a placeholder. The complete templates are reproduced in Appendix A.

An abbreviated view of the single-label prompt is shown in Listing 1.

```
<role>Expert Python Programming Assistant</role>

<task>
Identify the SINGLE most dominant logical error in the
provided Python code based on the taxonomy below.
</task>

<taxonomy>
  <category label="A">
    <title>Input</title>
    <description>Failing to receive all input values or
       using
    the incorrect data type for variables.</description>
  </category>
  ... [all 10 categories] ...
</taxonomy>

<output_rules>
- Output ONLY the single character label (A-J or N).
- STRICT: Any text other than the label is a failure.
</output_rules>

```

```
<code_to_analyze>
{code_input}
</code_to_analyze>
```

Listing 1: Single-label prompt template (abbreviated).

4.5 Evaluation Metrics

Standard binary accuracy is insufficient for multi-label evaluation because it does not account for partial correctness or hallucinations. In the definitions below, GT refers to the set of Ground Truth labels for a given snippet, and Pred refers to the set of labels predicted by the model. Four complementary cases were defined:

Table 4: The four evaluation cases used for scoring model predictions.

Case	Name	Definition
A	Precision (Subset)	$\text{Pred} \subseteq \text{GT}$. Did the model avoid all hallucinations? High score means high precision.
B	Recall (Superset)	$\text{GT} \subseteq \text{Pred}$. Did the model catch all real errors? High score means high recall.
C	Any Overlap	$\text{Pred} \cap \text{GT} \neq \emptyset$. Did the model get at least one label correct?
D	Jaccard Index	$\frlvert\text{Pred} \cap \text{GT}\frrvert / \frlvert\text{Pred} \cup \text{GT}\frrvert$. Penalises both omissions and hallucinations equally.

The **Jaccard Index** (Case D) is the primary accuracy metric, as it provides the most balanced view of performance. All scores are reported as percentages over the 100-question dataset.

4.6 Data Cleaning

A standardised cleaning pipeline was applied to all raw model outputs before evaluation:

- Responses of `"None"`, `"NaN"`, or empty strings were unified under the `"N"` label.
- Multi-label outputs were sorted and deduplicated so that `"J,I"` and `"I,J"` are treated identically.
- A regex-based extraction step parsed label characters from any surrounding explanatory text produced by a model despite the strict output rules.

5 Implementation

5.1 Repository Structure

The project is organised within the `Yuvraj/` folder as follows:

```
Yuvraj/
 analysis/
    benchmark_analysis.ipynb   # Analysis notebook
    benchmark_analysis.pdf     # Final findings PDF
 data/
    classification.pdf         # Taxonomy reference
    prompt.txt                 # Prompt templates
    yaksh100.json              # 100-question dataset
 results/
    csv/
       ground.csv             # Ground truth labels
       yaksh_single.csv       # Aggregated single-label
       yaksh_multi.csv        # Aggregated multi-label
    results_first_iteration/   # Raw outputs, Run 1
    results_second_iteration/  # Raw outputs, Run 2
 scripts/
     benchmark_yaksh.py         # Inference script
```

5.2 Benchmarking Script

The core of the implementation is `benchmark_yaksh.py`, a Python script that orchestrates the full inference pipeline:

1. Reads the dataset and prompt templates from disk.
2. Iterates over all questions and all configured models.
3. Constructs the user prompt by injecting the problem description and student code into the template.
4. Calls the OpenRouter API at temperature=0, storing raw responses incrementally.
5. Applies regex-based label extraction to obtain clean output.
6. Writes results to per-model text files (one line per question in the format `<id> <label(s)>`).
7. Supports **resumption**: already-processed question IDs are loaded on restart so a partial run can continue without re-querying completed items.
8. Supports a `--clean` flag to wipe previous results and start fresh.

A critical design decision was **incremental checkpoint saving**: each model-question pair is written to disk immediately after the API call, so a network interruption or credit exhaustion event does not lose prior work. This was essential given the large number of API calls required: 6 models $\times$ 100 questions $\times$ 2 modes $\times$ 2 runs = 2,400 individual queries.

5.3 Analysis Notebook

The analysis notebook, `benchmark_analysis.ipynb`, performs all post-processing and visualisation:

- Loads and cleans all raw model output files.
- Aggregates results into structured DataFrames.
- Computes the four evaluation metrics (Cases A through D) for each model, run, and mode.
- Generates pairwise agreement matrices across all model pairs.
- Produces radar charts showing per-category label distributions.
- Identifies behavioural profiles through label-count analysis.
- Exports summary statistics and visualisations.

6 Results and Discussion

6.1 Overview of Experimental Runs

Two full runs of the benchmark were conducted on the same 100-question dataset using identical prompts and zero-temperature settings. This two-run design allows measurement of *run-to-run stability*, an important but often overlooked dimension of LLM reliability.

6.2 Inter-Model Consensus

6.2.1 Overall Consensus Rates

A key finding is the dramatic collapse of inter-model agreement when the task shifts from single-label to multi-label classification:

Table 5: Overall consensus rates (all 6 models agree) across modes.

Mode	Run 1	Run 2
Single-Label	18.0% (18/100)	19.0% (19/100)
Multi-Label	9.0% (9/100)	8.0% (8/100)

Multi-label consensus is approximately half that of single-label consensus. This confirms that unconstrained error identification is substantially more subjective, and that models diverge significantly when allowed to assign multiple categories.

6.2.2 Closed-Source vs. Open-Source Group Agreement

Grouping the models by access type reveals a pronounced stability advantage for closed-source models:

- The **Closed-Source group** (GPT-5.2, Claude Sonnet 4.5, Gemini-2.5-flash) agreed with each other **38%** of the time in Single-Label mode (Run 1) and **40%** in Run 2.
- The **Open-Source group** (DeepSeek-v3.2, Qwen3-Coder, GPT-OSS-120b) showed lower cohesion at **25%** (Run 1) and **29%** (Run 2).

Closed-source models are therefore approximately 50% more consistent as a group than their open-source counterparts, a meaningful consideration for deployment in educational feedback pipelines. In Multi-Label mode, group cohesion collapses to roughly 12–14% for both groups, indicating that unconstrained classification removes any systematic advantage.

6.3 Pairwise Agreement Matrices

Pairwise agreement matrices were computed for all model pairs in both modes and both runs. Figure 1 shows all four matrices.

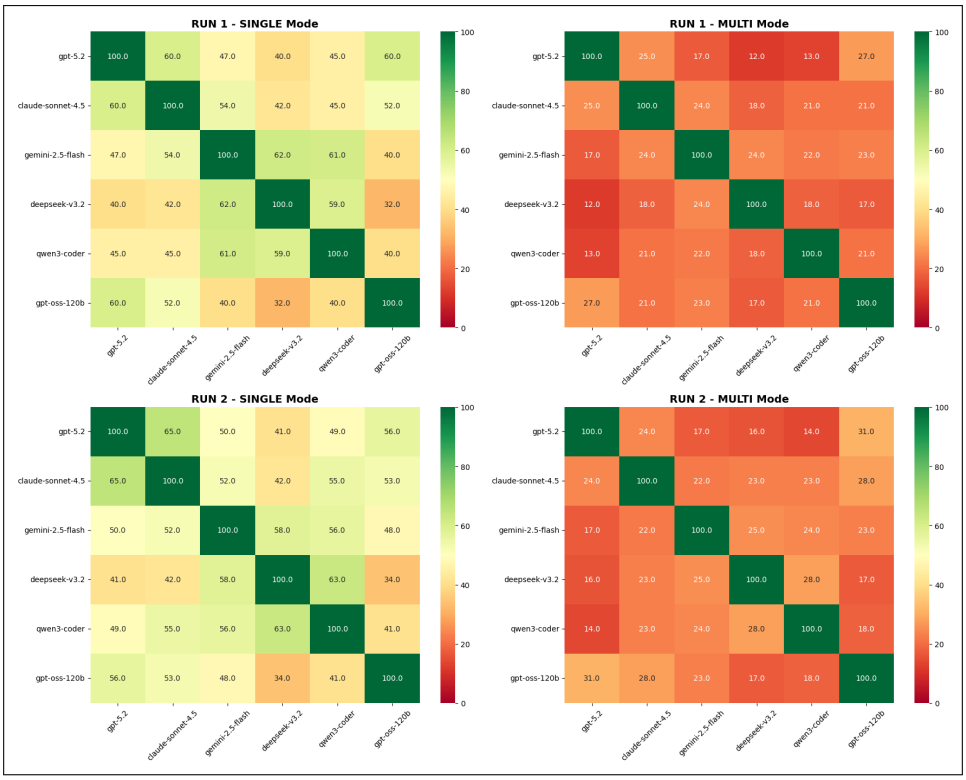

Figure 1: Pairwise agreement matrices for all six models across both runs and both modes.

Notable observations from the matrices:

- GPT-5.2 and Claude Sonnet 4.5 exhibit the highest pairwise agreement in Single-Label mode (60% in Run 1, 65% in Run 2), consistent with their shared closed-source profile.
- DeepSeek-v3.2 and GPT-OSS-120b show the lowest pairwise agreement (32% in Single-Label Run 1), reflecting high variance in open-source model outputs.
- In Multi-Label mode, all off-diagonal values collapse dramatically, with most pairs falling in the 12–28% range.

6.4 Accuracy Analysis

In Single-Label mode, where each model predicts exactly one error type, all models achieve relatively high Case A (Precision) scores, between 62% and 69%, because a single label is by definition a subset of any multi-label GT that contains it. However, Case B (Recall) scores are much lower (12–21%), indicating that a single

predicted label rarely covers all the true errors present in a snippet. Table 6 summarises single-mode performance averaged across both runs.

Table 6: Single-Label accuracy summary (averaged across Run 1 and Run 2, scores in %).

Model	Case A	Case B	Case C	Case D (Jaccard)
GPT-OSS-120b	66.5	21.0	66.5	**37.0**
Claude Sonnet 4.5	66.5	20.5	66.5	**36.4**
GPT-5.2	66.5	19.5	67.0	**36.2**
Gemini-2.5-Flash	65.5	16.5	65.5	**33.4**
Qwen3-Coder	65.0	16.0	65.0	**32.7**
DeepSeek-v3.2	65.0	14.5	65.0	**31.6**

Moving to the unconstrained Multi-Label mode reveals a more nuanced picture. Case A scores drop substantially (22–39%), reflecting the increased risk of hallucinating error categories that do not exist. Case B scores rise (33–55%), as models now assign multiple labels and are therefore more likely to include all true errors. The Jaccard Index, which penalises both omissions and false positives equally, provides the most balanced view; Table 7 presents these results averaged across both runs.

6.4.1 Multi-Label Leaderboard

Table 7: Multi-Label accuracy summary (averaged across Run 1 and Run 2, scores in %).

Model	Case A	Case B	Case C	Case D (Jaccard)
GPT-5.2	37.5	55.0	92.0	**57.1**
GPT-OSS-120b	28.5	52.5	84.0	**51.2**
Claude Sonnet 4.5	33.5	47.0	82.5	**49.4**
Gemini-2.5-Flash	37.0	33.5	78.0	**41.6**
Qwen3-Coder	22.0	45.0	84.0	**39.4**
DeepSeek-v3.2	28.0	43.0	78.0	**38.0**

GPT-5.2 is the clear leader in multi-label Jaccard accuracy (57.1%). GPT-OSS-120b is a notable runner-up (51.2%), outperforming expectations for an open-weights model. DeepSeek-v3.2 lags primarily due to over-flagging, i.e., hallucinating errors

that do not exist, which heavily penalises its Jaccard score. GPT-5.2 achieves the highest precision (Case A = 37.5%) among models with reasonable recall, reflecting its conservative prediction strategy.

Figure 2 visualises the Jaccard scores for both runs side by side, confirming the consistency of rankings across runs.

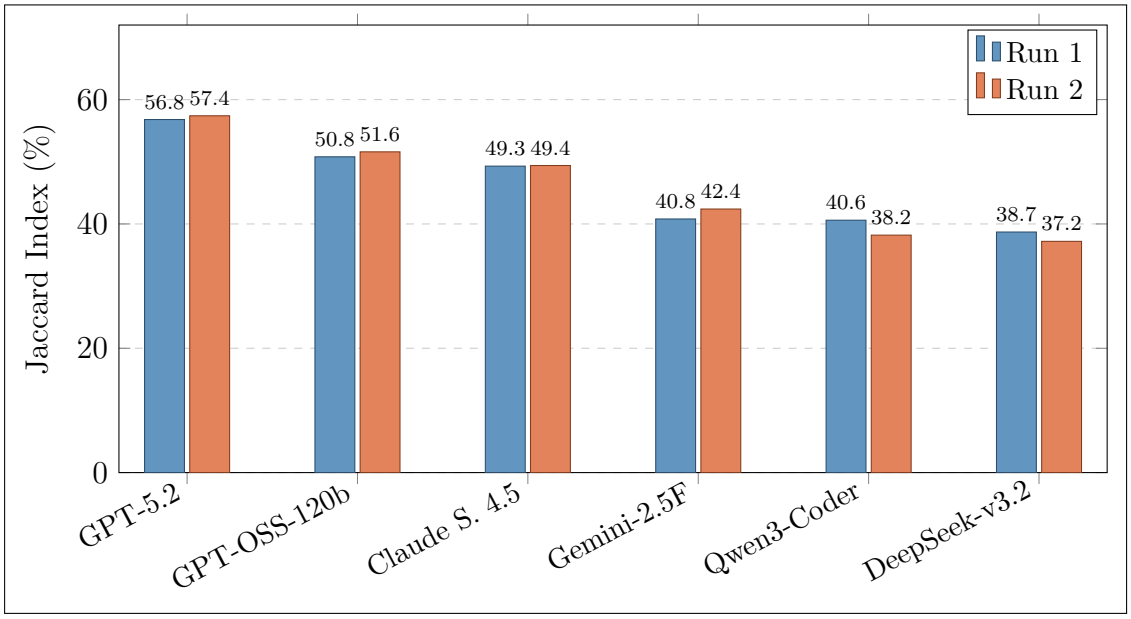

Figure 2: Multi-Label Jaccard Index scores per model for Run 1 and Run 2.

6.5 Behavioural Profiles: Conservative vs. Aggressive

A particularly informative finding is the identification of two distinct model behavioural profiles based on label-assignment patterns when switching from Single to Multi-Label mode. Figure 3 shows the retention rate (how often a model keeps its single-label answer when switching to multi-label mode) alongside the average number of labels assigned per snippet in multi-label mode.

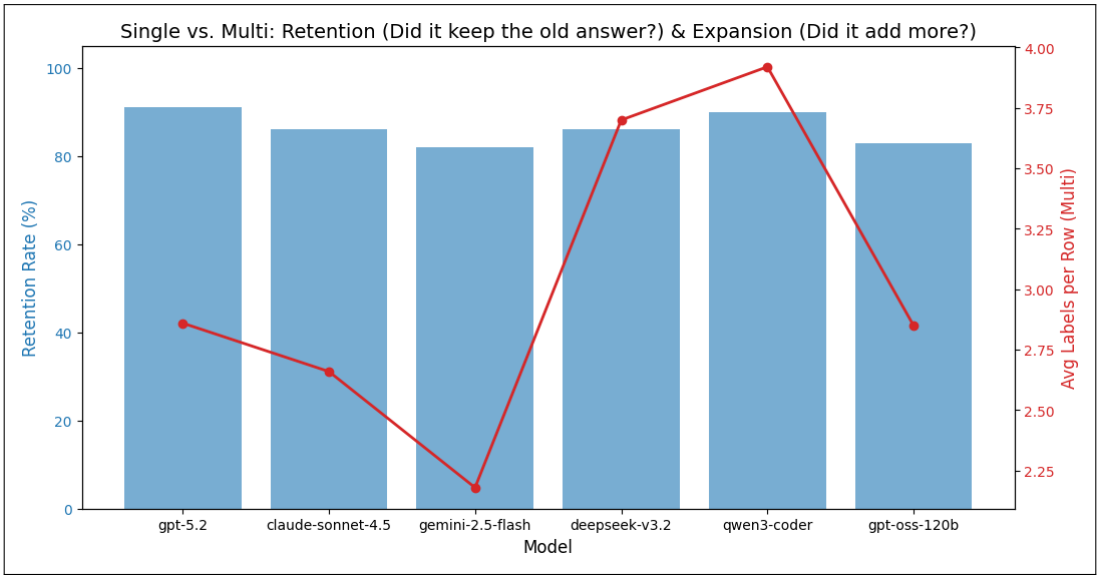

Figure 3: Retention vs. Expansion Across Models.

Conservative (Precision-focused): Gemini-2.5-flash adds the fewest labels per question in Multi-Label mode (average approximately 2.2 labels per snippet, the lowest of all models). It strongly prioritises avoiding false positives, resulting in high Case A scores but substantially lower recall.

Aggressive (Recall-focused): Qwen3-Coder and **DeepSeek-v3.2** adopt a broad identification strategy, assigning an average of approximately 3.8–4.0 labels per snippet. While this ensures high Case B recall, it results in significantly lower Jaccard accuracy due to the high false-positive rate.

This distinction has direct implications for deployment. A Conservative model suits scenarios where false alarms are costly (e.g., automatically blocking a submission), whereas an Aggressive model may be preferable when comprehensive coverage is the goal and a human reviewer will filter results.

6.6 Error Taxonomy Bias

All models except GPT-OSS-120b exhibit a strong bias toward identifying Category J (Conceptual) errors. This is the most abstract and broadly defined category, and LLMs appear systematically over-calibrated towards it. In Single-Label Run 1, Category J accounts for 33–44 of the 100 predictions for most models; the only exception is GPT-OSS-120b, which most frequently predicts Category I (Function).

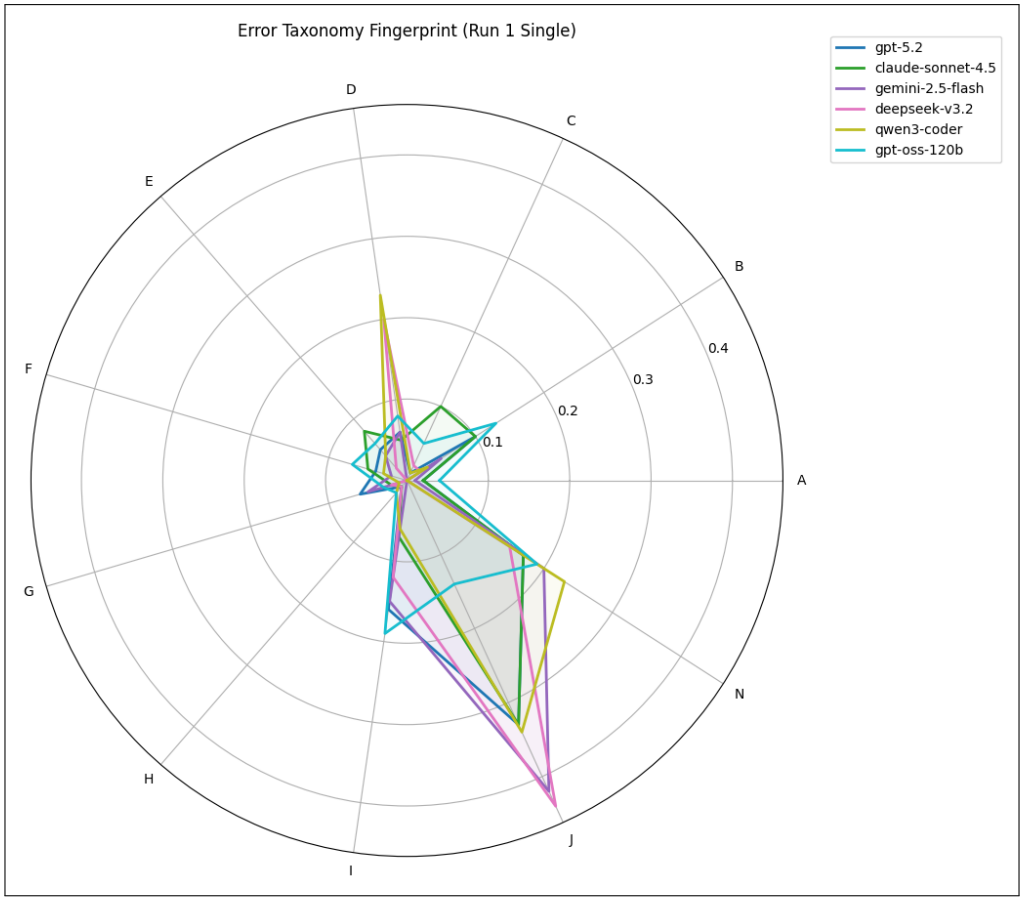

Figure 4: Error Taxonomy Fingerprint in Single Label mode (Run 1).

- **DeepSeek-v3.2** (Run 1 top-3: J=44, D=20, N=15) and **Qwen3-Coder** (J=34, N=23, D=23) show pronounced J-bias with a secondary preference for D (Computation), largely ignoring finer-grained structural error types.
- **GPT-OSS-120b** is the notable outlier: its top prediction is I (Function) with 19 occurrences, followed by N and J. This distinct fingerprint is clearly visible in the radar chart.
- **GPT-5.2** and **Claude Sonnet 4.5** exhibit a more balanced distribution, though J-bias is still present (J=33 each).

This suggests that current LLMs may be better suited to distinguishing *whether* a logical error exists at a high level, rather than pinpointing its specific structural category. Fine-grained categorisation across all ten types remains a significant challenge.

6.7 Subjectivity and the "Chaos Score"

For high-level, conceptually complex code samples, the six models produce maximally divergent diagnoses. In the dataset, four questions receive a "Chaos Score"

of 5/6, meaning five distinct diagnoses for the same snippet. A representative example is Row 28 ("Create a `LinearEquation` class"): GPT-5.2 predicts {I}, Claude predicts {C}, Gemini predicts {D}, DeepSeek predicts {J}, while Qwen and GPT-OSS both predict {B}.

6.8 Run-to-Run Stability

Despite zero-temperature settings, minor variations between Run 1 and Run 2 were observed for some models. These arose from non-determinism at the API infrastructure level, for example load balancing across model replicas. Overall Jaccard scores changed by at most 1.5 percentage points between runs for any model, and consensus rates were similarly stable. However, individual predictions on specific questions varied for roughly 5–8% of samples, underscoring the importance of multi-run evaluation when deploying LLMs in high-stakes educational contexts.

7 Conclusions and Future Work

7.1 Summary of Findings

This internship project produced a rigorous, reproducible benchmark for evaluating LLMs on logical error classification in student Python code. The following conclusions are supported by the data:

1. **Complexity breaks consensus.** Moving from Single to Multi-Label classification halves inter-model agreement (roughly 18% to 9%), demonstrating that unconstrained error identification is inherently more subjective.
2. **GPT-5.2 is the top multi-label performer.** With a Jaccard score of 57.1% averaged across both runs, it provides the best balance between catching real errors and avoiding hallucinated ones. Notably, GPT-OSS-120b leads in single-label Jaccard (37.0%), suggesting that open-weights models can be competitive in more constrained settings.
3. **Models have distinct behavioural profiles.** Gemini-2.5-flash prioritises precision (Conservative, approx. 2.2 labels/snippet), while Qwen3-Coder and DeepSeek-v3.2 prioritise recall (Aggressive, approx. 3.8–4.0 labels/snippet). The appropriate choice depends on the downstream use case.
4. **Logical error bias is near-universal.** Five of six models are over-calibrated toward Category J (Conceptual), often hallucinating conceptual flaws in otherwise correct code. GPT-OSS-120b is the exception, showing a distinct bias toward Category I (Function).
5. **Closed-source models are more stable as a group.** The closed-source trio agrees with each other approximately 38–40% of the time versus 25–29% for open-source models, making them more predictable for deployment.
6. **Human oversight remains essential.**. In approximately 4% of cases, model predictions exhibit near-zero agreement, indicating that fully automated logical error detection is not yet reliable without human review in educational settings.

7.2 Contributions

The concrete deliverables are:

- A structured, reusable benchmarking script (`benchmark_yaksh.py`) with resumption support, compatible with any set of OpenRouter-accessible models.
- Carefully engineered prompt templates for single-label and multi-label logical

error classification.

- Raw model outputs for two full runs across six models and two modes (2,400 API calls total).
- A complete analysis notebook (`benchmark_analysis.ipynb`) with reproducible plots, pairwise agreement matrices, and behavioural profile characterisation.
- A summary analysis report documenting all findings.

7.3 Limitations

- The dataset of 100 questions, while sufficient for initial analysis, is relatively small. Larger-scale evaluation would improve statistical power.
- The PG taxonomy may not cover all logical error types encountered in real student programs, particularly for more advanced exercises.
- API costs and rate limits constrained the total number of models and iterations that could be tested.

7.4 Future Work

Several directions are identified for extending this work:

1. **Larger datasets**: Expanding to several hundred or thousands of questions, ideally across multiple programming languages and difficulty levels.
2. **Few-shot and CoT prompting**: Integrating Chain-of-Thought and few-shot examples into prompts to improve fine-grained category discrimination.
3. **Fine-tuning**: Fine-tuning an open-weights model on annotated logical error data to reduce J-category bias.
4. **Integration with FOSSEE tools**: Embedding the best-performing model as an automated feedback module within Yaksh or a similar LMS platform.
5. **Inter-annotator agreement**: Developing a consensus annotation process to produce higher-quality ground truth labels for ambiguous cases.

8 Personal Reflection

This part-time internship provided a structured opportunity to combine several strands of learning, including prompt engineering, API-based software development, statistical analysis, and technical writing, into a cohesive research project. Below is a brief self-assessment of the key areas developed.

8.1 Technical Skills

- **API and SDK usage**: Gained hands-on experience designing and running large-scale inference pipelines via OpenRouter, including handling rate limits, authentication, and incremental result saving.
- **Prompt engineering**: Developed a deeper understanding of how taxonomy structure, output constraints, and role declarations affect LLM classification accuracy.
- **Data analysis in Python**: Strengthened proficiency in `pandas`, `numpy`, and `matplotlib`/`seaborn` for processing and visualising multi-label classification outputs.
- **Reproducible research practices**: Learned to structure a project for reproducibility, including clean file organisation, incremental checkpointing, and documentation.

8.2 Research Skills

Working within a research-oriented internship required developing comfort with ambiguity, particularly around ground-truth labelling and the inherent subjectivity of error classification. Designing evaluation metrics that could meaningfully capture partial correctness (Cases A through D) was a particularly instructive exercise.

References

[1] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[2] Tobias Kohn. The error behind the mistake: Identifying the misconception, not just the bug. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, pages 60–69. ACM, 2017.

[3] Yanggyu Lee, Suchae Jeong, and Jihie Kim. Improving LLM classification of logical errors by integrating error relationship into prompts. *arXiv preprint arXiv:2404.19336*, 2024.

[4] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837, 2022.

[5] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213, 2022.

[6] OpenAI. OpenAI models overview. https://platform.openai.com/docs/models, 2025.

[7] Anthropic. Claude models overview. https://docs.anthropic.com/en/docs/about-claude/models/overview, 2025.

[8] Google. Gemini API models. https://ai.google.dev/gemini-api/docs/models, 2025.

[9] DeepSeek-AI. DeepSeek-V3 technical report. https://github.com/deepseek-ai/DeepSeek-V3, 2025.

[10] Alibaba Cloud. Qwen3 technical report. https://qwen.ai/blog?id=qwen3, 2025.

[11] OpenRouter. OpenRouter API documentation. https://openrouter.ai/docs, 2025.

[12] FOSSEE, IIT Bombay. Yaksh: Open-source online test platform. https://yaksh.fossee.in, 2025.

A Prompt Templates

A.1 Single-Label Prompt (PROMPT_ITER1_SINGLE)

```
<role>Expert Python Programming Assistant</role>

<task>
Identify the SINGLE most dominant logical error in the
provided Python code based on the taxonomy below.
</task>

<taxonomy>
  <category label="A">
    <title>Input</title>
    <description>Failing to receive all input values or
    using the incorrect data type for variables.
    </description>
  </category>
  <category label="B">
    <title>Output</title>
    <description>Non-compliance with required formats or
    outputting incorrect string literals.</description>
  </category>
  <category label="C">
    <title>Variable</title>
    <description>Storing incorrect values or specifying
    the wrong data type for a variable.</description>
  </category>
  <category label="D">
    <title>Computation</title>
    <description>Calculating with incorrect values or
    using the wrong operations.</description>
  </category>
  <category label="E">
    <title>Condition</title>
    <description>Incorrect or insufficient conditional
    operations in declarations.</description>
  </category>
  <category label="F">
    <title>Branching</title>
```

```
    <description>Incorrect program flow (e.g., wrong break
    placement or if-if instead of if-else).</description>
  </category>
  <category label="G">
    <title>Loop</title>
    <description>Incorrect conditions or variables
    used in loop declarations.</description>
  </category>
  <category label="H">
    <title>Array/String</title>
    <description>Incorrect initialization or referencing
    an incorrect index.</description>
  </category>
  <category label="I">
    <title>Function</title>
    <description>Incorrectly defined parameters/return
    values or wrong arguments in calls.</description>
  </category>
  <category label="J">
    <title>Conceptual</title>
    <description>Solving the wrong problem or missing
    necessary loops/conditions entirely.</description>
  </category>
  <category label="N">
    <title>No Error</title>
    <description>No logical error present.</description>
  </category>
</taxonomy>

<instructions>
1. Analyse the code inside the <code_to_analyze> tags.
2. If multiple errors exist, select the one that most
   fundamentally causes the code to fail.
</instructions>

<output_rules>
- Output ONLY the single character label
  (A, B, C, D, E, F, G, H, I, J, N).
- Do NOT include the category title, explanations,
```

```
  or any punctuation.
- STRICT: Any text other than the label is a failure.
</output_rules>

<code_to_analyze>
{code_input}
</code_to_analyze>
```

Listing 2: Full single-label prompt template.

A.2 Multi-Label Prompt (PROMPT_ITER_MULTI)

```
<role>Expert Python Programming Assistant</role>

<task>
Identify ALL applicable logical error types in the
provided Python code based on the taxonomy below.
</task>

<taxonomy>
  <category label="A">
    <title>Input</title>
    <description>Failing to receive all input values or
    using the incorrect data type for variables.
    </description>
  </category>
  <category label="B">
    <title>Output</title>
    <description>Non-compliance with required formats or
    outputting incorrect string literals.</description>
  </category>
  <category label="C">
    <title>Variable</title>
    <description>Storing incorrect values or specifying
    the wrong data type for a variable.</description>
  </category>
  <category label="D">
    <title>Computation</title>
    <description>Calculating with incorrect values or
    using the wrong operations.</description>
  </category>
```

```
  <category label="E">
    <title>Condition</title>
    <description>Incorrect or insufficient conditional
    operations in declarations.</description>
  </category>
  <category label="F">
    <title>Branching</title>
    <description>Incorrect program flow (e.g., wrong break
    placement or if-if instead of if-else).</description>
  </category>
  <category label="G">
    <title>Loop</title>
    <description>Incorrect conditions or variables
    used in loop declarations.</description>
  </category>
  <category label="H">
    <title>Array/String</title>
    <description>Incorrect initialization or referencing
    an incorrect index.</description>
  </category>
  <category label="I">
    <title>Function</title>
    <description>Incorrectly defined parameters/return
    values or wrong arguments in calls.</description>
  </category>
  <category label="J">
    <title>Conceptual</title>
    <description>Solving the wrong problem or missing
    necessary loops/conditions entirely.</description>
  </category>
  <category label="N">
    <title>No Error</title>
    <description>No logical error present.</description>
  </category>
</taxonomy>

<instructions>
1. Evaluate the code inside the <code_to_analyze> tags
   against every category in the taxonomy.
```

```
2. Select all labels that apply to the logic of the code.
</instructions>

<output_rules>
- Output labels as a comma-separated list (e.g., A,C,E).
- If N is chosen, it must be the ONLY label in the output.
- Do NOT include spaces, titles, or descriptions.
- STRICT: No conversational filler or explanations.
</output_rules>

<code_to_analyze>
{code_input}
</code_to_analyze>
```

Listing 3: Full multi-label prompt template.

B Ground Truth Label Distribution

The ground truth (GT) labels in `ground.csv` show the following category distribution across the 100 questions. Counts represent the number of questions containing each label. Because the data are multi-label, the totals sum to more than 100. Blank entries and entries with no labels are counted as "No Error" (N).

Table 8: Ground Truth category distribution from `ground.csv`.

Category	ID	Count	Percent
Conceptual	J	48	48%
Function	I	43	43%
Output	B	44	44%
Computation	D	37	37%
Input	A	18	18%
Condition	E	11	11%
Loop	G	7	7%
Variable	C	3	3%
Branching	F	0	0%
Array/String	H	9	9%
No Error	N	19	19%

C Raw Accuracy Data

For full reproducibility, Tables 9 and 10 present the per-run accuracy values from which the averages in the main paper are derived.

Table 9: Single-Label raw accuracy per run (scores in %).

	Run 1				**Run 2**			
Model	A	B	C	D	A	B	C	D
GPT-5.2	67	21	68	37.3	66	18	66	35.1
Claude Sonnet 4.5	65	20	65	35.8	68	21	68	36.9
Gemini 2.5 Flash	66	18	66	34.6	65	15	65	32.1
DeepSeek-v3.2	68	17	68	34.4	62	12	62	28.8
Qwen3-Coder	65	15	65	32.2	65	17	65	33.2
GPT-OSS-120b	64	21	64	36.0	69	21	69	38.0

Table 10: Multi-Label raw accuracy per run (scores in %).

	Run 1				**Run 2**			
Model	A	B	C	D	A	B	C	D
GPT-5.2	39	55	92	56.8	36	55	92	57.4
Claude Sonnet 4.5	35	48	83	49.3	32	46	82	49.4
Gemini 2.5 Flash	36	33	77	40.8	38	34	79	42.4
DeepSeek-v3.2	30	46	78	38.7	26	40	78	37.2
Qwen3-Coder	22	45	86	40.6	22	45	82	38.2
GPT-OSS-120b	29	53	84	50.8	28	52	84	51.6