Semester Long Internship Report

On

A Multi-Stage LLM Pipeline for Reliable Python Syntax Error Explanations

Submitted by

H Priyanka

GitHub Repository

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

Under the guidance of

Dr. Prabhu Ramachandran

FOSSEE Team, IIT Bombay

Free/Libre and Open Source Software for Education

Indian Institute of Technology Bombay

March, 2026

Acknowledgement

I would like to express my sincere gratitude to the FOSSEE team at IIT Bombay for providing me with the opportunity to learn, implement, and understand the applications of technology in a real-world, large-scale project. This internship allowed me to explore the intersection of Large Language Models, programming education, and structured AI pipeline design, making the experience both meaningful and impactful.

I am thankful for the access to the Yaksh online programming platform dataset, which formed the empirical foundation of this project. Working with authentic student submissions made the research grounded and practically relevant, while also helping me better understand real-world programming challenges faced by learners.

I would like to express my sincere appreciation to my mentor, **Dr. Kushal Shah**, for his continuous guidance and support throughout the internship. His insights and feedback played a crucial role in shaping the direction of this work and helped me navigate challenges effectively. The experience of working under his mentorship made this journey more structured, insightful, and rewarding. I would also like to acknowledge **Dr. Prabhu Ramachandran** for his valuable feedback and guidance.

Finally, I am grateful to everyone who supported me during this journey. This internship has been a valuable learning experience, and I look forward to applying these learnings in future work and contributing to impactful, real-world software development.

Contents

Acknowledgement **1**

1 Introduction **3**

1.1 About FOSSEE and the Yaksh Platform 3

1.2 About the Project 3

1.3 Motivation 4

1.4 Objectives 5

1.5 Technologies and Tools Used 5

2 Problem Statement **6**

3 Work Done / Implementation **8**

3.1 Dataset Curation 8

3.2 Single-LLM Baseline Pipeline 9

3.3 Multi-Stage Multi-LLM Pipeline 10

3.3.1 Stage A: Independent Patch Proposals 10

3.3.2 Stage B: Cross-Model Critique 11

3.3.3 Stage C: Judge Synthesis 11

3.4 Output Schema and Data Management 12

3.5 Evaluation Methodology 13

4 Conclusion and Future Scope **14**

4.1 Results Summary 14

4.2 Conclusion 15

4.3 Limitations 16

4.4 Future Scope 16

References **18**

1 Introduction

1.1 About FOSSEE and the Yaksh Platform

FOSSEE (Free/Libre and Open Source Software for Education) is an initiative under the National Mission on Education through ICT (NME-ICT), supported by the Ministry of Education, Government of India, and hosted at IIT Bombay. Its primary goal is to promote the adoption of open-source software tools in academic institutions, spanning domains such as scientific computing, circuit simulation, and programming education.

As part of this initiative, FOSSEE maintains the Yaksh online programming portal a platform used across introductory Python programming courses. The Yaksh corpus contains approximately 2.81 million student submissions spanning 398 unique programming questions. This rich dataset of authentic student code and interpreter feedback provides an exceptional resource for studying automated programming assistance at scale.

1.2 About the Project

This internship project focuses on developing and evaluating a Multi-Stage Large Language Model (LLM) Pipeline for generating reliable explanations of Python syntax errors. Syntax errors are among the most common and disruptive problems encountered by novice programmers. While Python's interpreter points to where a parse failure occurs, it rarely explains the underlying syntactic rule that was violated leaving students uncertain about both the cause and the appropriate corrective action.

Large Language Models have demonstrated the ability to generate natural language explanations resembling instructor feedback. However, single-pass LLM explanations are not consistently reliable models can hallucinate incorrect causes, miss downstream error effects, or produce explanations poorly aligned with the actual interpreter feedback.

To address this, the project implements and evaluates a structured three-stage pipeline, illustrated in Figure 1:

1. Stage A - Independent candidate generation by two proposer LLMs (Qwen3-

Coder and GPT-OSS-120B)

2. Stage B - Cross-model critique grounded in interpreter feedback
3. Stage C - Judge-based synthesis by DeepSeek-V3.2 to produce a final, consolidated explanation

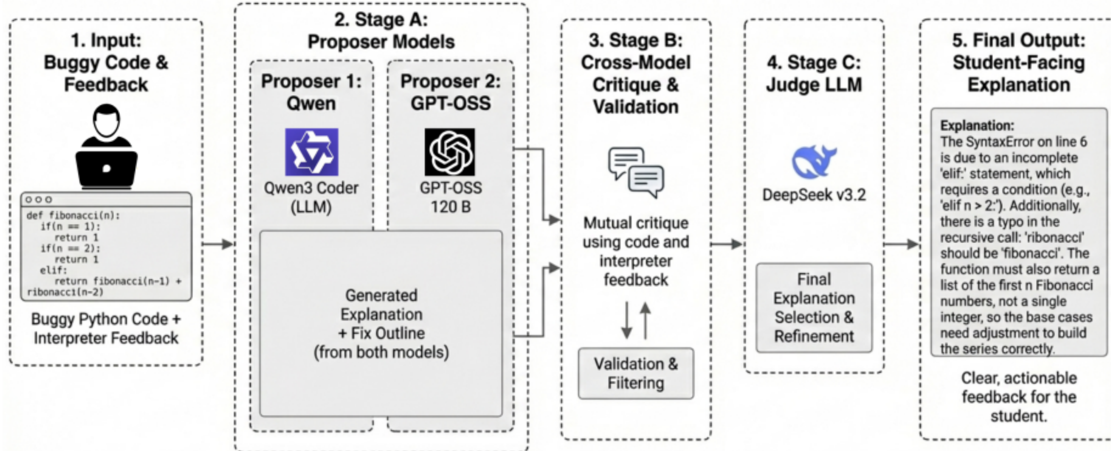

Figure 1: Overview of the proposed multi-stage pipeline for Python syntax error explanation generation, showing independent proposer LLMs (Stage A), cross-model critique grounded in interpreter feedback (Stage B), and final judge-based synthesis (Stage C) to produce a reliable, evidence-aligned student-facing explanation.

1.3 Motivation

Syntax errors such as missing colons, incorrect indentation, or malformed expressions are frequent in introductory programming. The Python interpreter's feedback is typically terse and points to the line where parsing fails, not necessarily where the mistake was made. For novice learners, this gap between the error message and the actual conceptual mistake can be deeply confusing and demoralising.

Automated explanation systems powered by LLMs offer a scalable solution. However, deploying them in educational settings demands a higher standard of reliability than in general use. An incorrect explanation in a classroom context can actively mislead a student and impede conceptual understanding. This motivates a pipeline that introduces explicit verification and synthesis, rather than relying on a single model's unverified output.

1.4 Objectives

The primary objectives of this internship project were:

- To curate a high-quality benchmark of 100 authentic Python syntax-error programs from the Yaksh corpus using a reproducible filtering and deduplication pipeline
- To implement a single-LLM baseline pipeline as a control for comparison
- To design and implement a three-stage multi-LLM pipeline with independent proposals, cross-model critique, and judge-based synthesis
- To evaluate and compare both pipelines through manual preference analysis across all 100 cases
- To analyse structural properties of generated explanations and the behaviour of the judge model across stages
- To document findings and contribute the pipeline as an extensible open-source framework for future research

1.5 Technologies and Tools Used

- Language: Python 3
- LLM Models: Qwen3-Coder (Proposer 1), GPT-OSS-120B (Proposer 2), DeepSeek-V3.2 (Judge)
- Libraries: openai, json, openpyxl, pandas, re, hashlib, Python built-in compile()
- Data Format: Structured JSON schemas for all pipeline outputs; Excel (.xlsx) for storage and analysis
- Tools: Jupyter Notebook, Git, GitHub
- Dataset: Yaksh Online Programming Portal (IIT Bombay)

2 Problem Statement

Debugging Python code is a foundational skill in programming education, yet it remains one of the most frustrating aspects of the learning process for beginners. When a student's code contains a syntax error, the Python interpreter generates a traceback identifying the line at which parsing failed but provides no explanation of the underlying syntactic rule that was violated, nor any pedagogically useful guidance on how to correct the mistake.

Students are consequently left to interpret messages such as `SyntaxError: invalid syntax`, `IndentationError: unexpected indent`, or `TabError: inconsistent use of tabs and spaces` with little context. This gap creates a significant barrier, especially in large-scale programming courses where individual instructor support is limited.

The specific requirements this project aims to address are:

1. **Reliable Error Explanation:**
   - Correctly identify the syntactic rule that was violated, not just the line where parsing failed
   - Align the explanation with the actual interpreter feedback provided
   - Avoid hallucinated or plausible-but-incorrect causes
2. **Pedagogically Appropriate Guidance:**
   - Provide a clear, step-by-step fix outline that guides the student without revealing the complete corrected solution
   - Address downstream effects of the syntax error (e.g., variable name mismatches, invalid operations on wrongly-typed objects)
3. **Structured Verification and Auditability:**
   - Make the reasoning process explicit and inspectable, rather than collapsing hypothesis generation, verification, and resolution into a single unverifiable step
   - Produce outputs in structured formats supporting automatic parsing and metric computation

Existing single-pass LLM approaches implicitly merge all of these steps into a single model call, providing no mechanism to detect or correct errors in reasoning mid-generation. The result is an output that may be fluent and plausible yet incorrect particularly problematic in educational settings where trust in the system directly affects student learning outcomes.

This project therefore proposes a structured multi-stage pipeline that explicitly separates hypothesis generation, verification through critique, and evidence-grounded synthesis enabling each stage to refine and correct the outputs of previous stages before a final explanation is committed.

3 Work Done / Implementation

3.1 Dataset Curation

The first major task was constructing a high-quality benchmark of Python syntax-error programs from the Yaksh corpus. The Yaksh dataset spans approximately 2.81 million submissions across 398 programming questions. Table 1 shows the distribution of execution feedback categories across the full corpus.

Table 1: Distribution of major execution error categories in the Yaksh dataset

Error Category	Count
WrongAnswer	401,285
SyntaxError	372,906
NameError	142,148
TypeError	92,294
AttributeError	34,456
ValueError	30,033
AssertionError	18,893
TimeLimitExceeded	25,001
Other Runtime Errors	28,348

The benchmark construction followed a deterministic, five-step filtering pipeline:

1. **Explicit Syntax Error Detection:** Submissions were scanned for known Python parser messages (`SyntaxError`, `IndentationError`, `TabError`) using pattern matching.
2. **Compile-Time Fallback:** For submissions with empty execution feedback, code was re-evaluated using Python's built-in `compile()` function to recover latent parse-time failures.
3. **Filtering Non-Informative Code:** Trivial submissions empty programs, single identifiers, or bare function calls were excluded. Only submissions with meaningful program structure were retained.
4. **Deduplication via Structural Normalisation:** Code was normalised by abstracting identifiers, literals, and formatting, then hashed. Submissions with identical hashes were treated as duplicates, retaining only one per hash.

5. **Diversity Control Across Questions:** At most one submission per programming question was selected in the primary pass to avoid over-representation of frequently attempted problems.

The final benchmark consists of 100 unique, authentic parse-time failure programs, comprising 64 miscellaneous explicit syntax failures, 24 `SyntaxError` cases, and 12 `IndentationError` cases, all produced by the Python interpreter.

3.2 Single-LLM Baseline Pipeline

To establish a control for comparison, a single-LLM baseline was implemented where each of the 100 questions is processed independently by a single model, prompted with the buggy code and interpreter feedback.

The model produces a structured JSON output containing:

- `helpful_output` - natural language explanation of the error
- `fix_outline` - ordered list of corrective steps
- `confidence` - self-reported confidence score

This standard "prompt $\rightarrow$ response" approach serves as the baseline against which the multi-stage pipeline is compared. Figure 2 shows a representative example of how a single-LLM explanation correctly identifies the immediate syntax error at `a=array[]` but fails to account for downstream inconsistencies such as the variable name mismatch (`a` vs. `arr`) or the invalid `.shape` assignment.

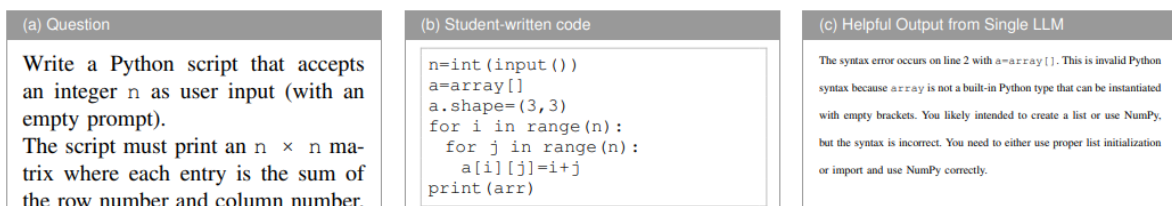

Figure 2: Example syntax-error case showing the problem description, student-written buggy Python code, and the explanation generated by a single LLM. Single-pass explanations identify the immediate syntax error but often fail to account for downstream inconsistencies or secondary issues in the code.

3.3 Multi-Stage Multi-LLM Pipeline

The core contribution of this internship is a three-stage verification pipeline that converts latent model uncertainty into explicit, inspectable critique signals, resolved through a dedicated judge model. The complete stage-wise outputs for the same example case are shown in Figure 3.

3.3.1 Stage A: Independent Patch Proposals

In Stage A, Qwen3-Coder and GPT-OSS-120B are queried independently using the same input. The models do not share context, ensuring that agreement between them emerges through independent reasoning paths.

Each model produces a structured JSON output with:

- `helpful_output` - natural language explanation
- `fix_outline` - ordered list of corrective steps
- `fixed_code` - proposed corrected code (for internal pipeline use)
- `confidence` - self-reported confidence (0–1)

Across 100 questions, this stage produced 200 proposals (100 per model), stored in `stageA_proposals.xlsx`. A representative example output is shown below:

```
{
"helpful_output": "The syntax error indicates Python doesn’t recognize
 the function declaration format you’re using. This appears to be
 C-style syntax which is not valid in Python. Python uses a different
 function definition syntax with the ’def’ keyword and doesn’t require
 type declarations for parameters or return values.",
"fix_outline": [
 "Replace the C-style function declaration with Python’s ’def’ syntax",
 "Remove type declarations (int) from parameters and function signature",
 "Replace curly braces with proper Python indentation",
 "Add a colon after the function definition line"
],
"confidence": 0.95
}
```

Listing 3.1: Stage A: Qwen3-Coder proposal for a C-style function syntax error

3.3.2 Stage B: Cross-Model Critique

In Stage B, each model critiques the proposal generated by the other model, using only the original buggy code and interpreter feedback as evidence. This externalises implicit disagreement into structured, inspectable artefacts.

Each critique is a structured JSON object containing:

- `evidence_alignment` - pass/fail alignment with interpreter output
- `helpfulness` - pass/fail assessment of pedagogical utility
- `issues` - list of identified problems, each with severity (high/med/low) and supporting evidence
- `suggested_improvement` - concrete suggestion for refinement

This stage generated 200 critiques across the 100 questions, stored in `stageB_critiques.xlsx`. An example critique is shown below:

```
{
"evidence_alignment": "pass",
"helpfulness": "pass",
"issues": [
 {
  "severity": "med",
  "issue": "Did not mention the missing colon after function header",
  "evidence": "Python requires ':' after the parameter list"
 }
],
"suggested_improvement": "Add a note about the required ':' after
 the function definition line to fully address the syntax error."
}
```

Listing 3.2: Stage B: GPT-OSS critique of Qwen's proposal

3.3.3 Stage C: Judge Synthesis

In Stage C, DeepSeek-V3.2 receives all prior inputs and reasons over all available evidence to either select the proposal that best survives critique, or produce a hybrid fix combining the strongest elements of both proposals.

The final output contains:

- `final_helpful_output` - consolidated, evidence-aligned explanation
- `final_fix_outline` - final step-by-step corrective guidance
- `chosen_source` - `hybrid`, `gpt_oss`, or `qwen`
- `confidence` - judge's confidence score

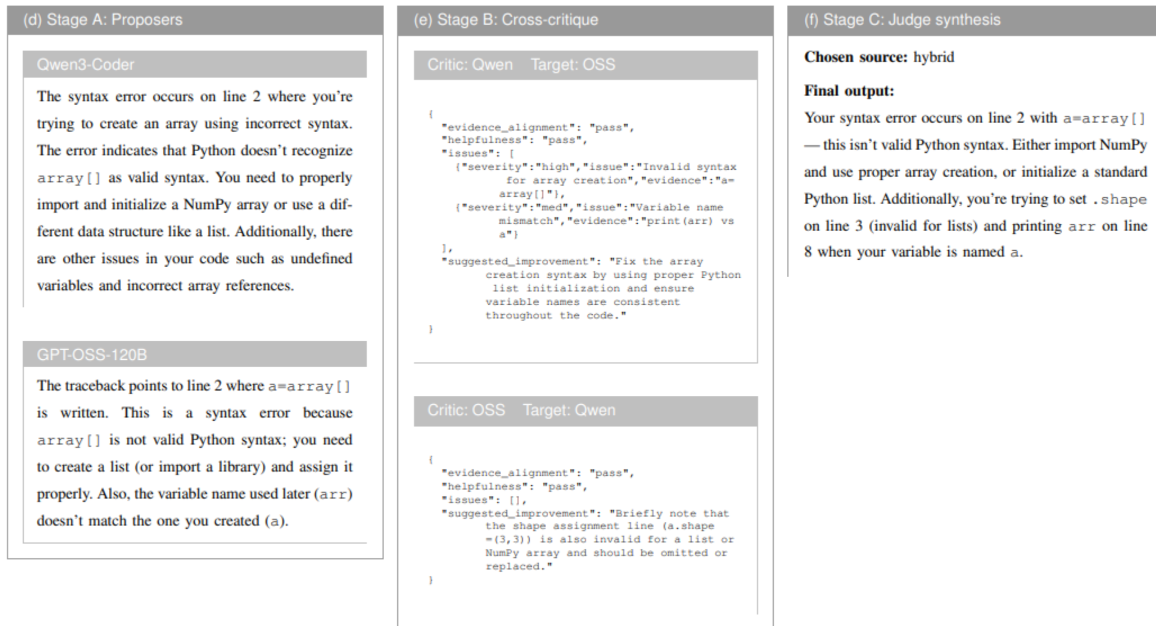

Figure 3: Stage-wise outputs of the multi-stage pipeline for the same syntax-error case: independent proposer explanations (Stage A, left), cross-model critiques grounded in interpreter feedback (Stage B, centre), and judge-based synthesis (Stage C, right) resolving disagreements to produce a more complete and evidence-aligned explanation.

Across 100 questions, the judge selected hybrid explanations in 67 cases, GPT-OSS in 29 cases, and Qwen in 3 cases, with an average confidence of 0.982. Results are stored in `stageC_final_helpful_output.xlsx`.

3.4 Output Schema and Data Management

All outputs across the three stages were constrained to strict JSON schemas, enabling:

- Automatic parsing and validation
- Metric computation across stages
- Auditable, side-by-side comparison of model outputs

Malformed or missing outputs were explicitly tracked as parse failures rather than silently discarded, ensuring full accountability across the evaluation.

3.5 Evaluation Methodology

Explanation quality was evaluated through manual comparative analysis between the single-LLM baseline and the multi-stage pipeline output across all 100 syntax-error cases. Each case was independently labelled as:

- M - Multi-stage preferred
- T - Tie (both equally useful)
- S - Single-model preferred

Labelling criteria included correctness of error attribution, alignment with interpreter feedback, completeness of corrective guidance, and clarity for novice programmers.

Additionally, fix-outline length (number of explicit corrective steps) was used as a coarse structural measure to assess how explicitly each pipeline decomposes corrective reasoning. Explanations were grouped into ≤ 1 step, 2 steps, or ≥ 3 steps.

4 Conclusion and Future Scope

4.1 Results Summary

The evaluation results across 100 syntax-error cases are summarised in Table 2.

Table 2: Manual preference outcomes across 100 syntax-error cases

Outcome	Count	Percentage
Multi-stage preferred (M)	69	69%
Tie (T)	29	29%
Single-model preferred (S)	2	2%

The multi-stage pipeline is preferred in 69% of cases and is at least as good as the single-model baseline in 98% of cases. Figure 4 shows the distribution of fix-outline lengths for both pipelines, demonstrating that the multi-stage pipeline more consistently produces structured, multi-step explanations that decompose corrective reasoning clearly without prematurely revealing complete solutions.

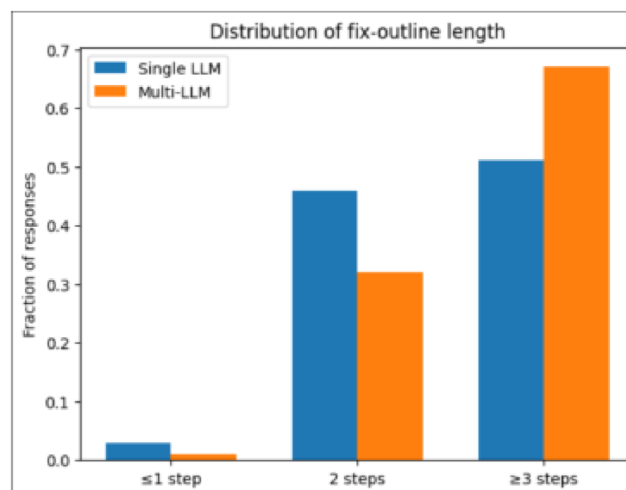

Figure 4: Distribution of fix-outline length for single-model and multi-stage explanations. Fix-outline length denotes the number of explicit corrective steps in an explanation. The multi-stage pipeline more often produces structured, multi-step reasoning (≥ 3 steps), while single-model explanations more frequently produce only 2 steps.

The rare instances where single-model output was preferred (2%) correspond primarily to parser ambiguity cases where interpreter feedback was underspecified these do not indicate a systematic weakness in the pipeline.

Figure 5 shows the judge model's choice distribution across the 100 cases. Hybrid explanations were produced in 67 cases (67%), indicating that candidate proposals are typically complementary rather than redundant, and that synthesising them produces stronger outputs than selecting either alone.

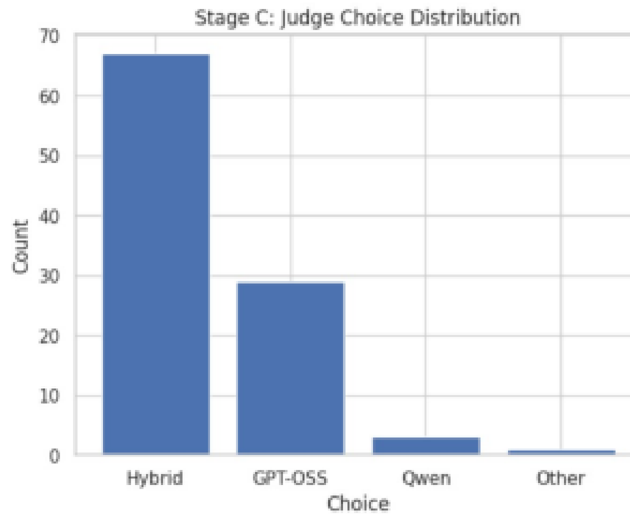

Figure 5: Distribution of judge model choices (DeepSeek-V3.2) across 100 syntax-error cases. The judge most frequently produces hybrid explanations (67 cases) by combining complementary, evidence-aligned components from both proposers. GPT-OSS is selected outright in 29 cases, Qwen in 3, and other in 1, corresponding to edge cases with parser ambiguity or underspecified error messages.

4.2 Conclusion

This internship successfully designed, implemented, and evaluated a structured multi-stage LLM pipeline for generating reliable Python syntax error explanations. By decomposing explanation generation into independent proposal, cross-model critique, and judge-based synthesis, the pipeline converts model disagreement from a source of error into a mechanism for improving reliability and transparency.

Key outcomes achieved during this internship include:

- A curated, reproducible benchmark of 100 authentic Python syntax-error programs from the Yaksh corpus
- A fully implemented and evaluated single-LLM baseline pipeline
- A three-stage multi-LLM pipeline producing structured, auditable outputs across all stages (`stageA_proposals.xlsx`, `stageB_critiques.xlsx`, `stageC_final_helpful_output.xlsx`)

- Manual comparative evaluation demonstrating substantial and consistent improvement over the single-model baseline (preferred in 69% of cases, at least as good in 98%)
- An open-source repository and accompanying research paper submitted for peer review

4.3 Limitations

- Evaluation was conducted through manual comparative analysis, which involves inherent subjectivity and does not scale to larger datasets without automated metrics
- In cases of parser ambiguity or highly atypical error messages, the multi-stage pipeline tends to produce conservative explanations, occasionally less decisive than a single-model call
- The benchmark is limited to 100 cases from introductory Python courses; generalisation to advanced programs or other languages remains untested
- The pipeline incurs higher computational cost and latency than the single-model baseline due to multi-model invocations across three stages

4.4 Future Scope

- Automated Evaluation Metrics: Develop or adapt automated metrics (e.g., alignment scoring with interpreter evidence, fix-step precision and recall) to enable large-scale evaluation beyond 100 cases
- Extension to Runtime Errors: Extend the pipeline to runtime errors such as `NameError`, `TypeError`, and `AttributeError`, which would significantly broaden its educational impact
- Student Study: Conduct a controlled user study with novice programmers to measure whether multi-stage explanations improve error resolution speed and conceptual understanding compared to single-model explanations
- Integration into Yaksh: Integrate the pipeline as a live feedback module within the Yaksh platform, enabling real-time LLM-assisted explanations for student

submissions

- Continuous Learning: Implement a feedback loop where student ratings of explanations are used to fine-tune or re-rank model outputs over time
- Broader Language Support: Adapt the pipeline architecture for other programming languages used in educational settings (e.g., C, Java, JavaScript)

References

[1] T. Phung et al., “Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models,” *arXiv.org*, Jan. 24, 2023. https://arxiv.org/abs/2302.04662

[2] Qwen Team, “Qwen3-Coder-Next,” https://huggingface.co/Qwen/Qwen3-Coder-Next

[3] DeepSeek AI, “DeepSeek-V3.2,” https://huggingface.co/deepseek-ai/DeepSeek-V3.2

[4] OpenAI, “GPT-OSS-120B,” https://huggingface.co/openai/gpt-oss-120b

[5] https://github.com/yxpx/fossee/tree/main, 2026.