Semester Long Internship - Autumn 2025

Duration: 28 October 2025 – 15 April 2026

Project Report

On

LLM-Based Educational Tutoring Systems

Error Analysis, Hint Generation, and Adaptive Question Sequencing

Submitted by:

Abhishek Balaji Chavan

Vidyalanakar Institute of Technology, Mumbai, India

Under the guidance of:

Dr. Kushal Shah
Prof. Prabhu Ramachandran

April 15, 2026

Acknowledgements

I would like to express my sincere gratitude to the FOSSEE team at IIT Bombay for providing me with the opportunity to work on this internship project. I am especially thankful to my supervisor Dr. Kushal Shah and Prof. Prabhu Ramachandran for their continuous guidance, valuable feedback, and encouragement throughout the internship period.

I would also like to thank the entire FOSSEE development team for their support and for providing a practical environment in which to work on AI-assisted programming education. The internship gave me direct experience with natural language processing, large language models, and software engineering in an educational setting.

Finally, I thank my family and friends for their unwavering support during the course of this internship.

Abstract

This report presents the work completed during a six-month internship at FOSSEE, focused on building tutoring support for Python programming education using large language models (LLMs). The work spans three connected tasks: classifying student errors, generating hints that do not reveal the solution, and sequencing questions on the basis of student performance.

In the **first task**, we developed an LLM response analyzer that evaluates AI-generated tutoring hints across six analytical dimensions: AST-based code validation, Socratic questioning detection, educational tone classification, error-specific relevance scoring against nine Python exception types, readability metrics, and contextual analysis. Building on this analysis infrastructure, we designed a three-stage hint generation pipeline that separates error analysis at temperature 0.2, pedagogical strategy selection at temperature 0.3, and student-facing hint generation at temperature 0.5. The archived five-case pipeline run completed without API or JSON-parsing failure and averaged 1,805 tokens per hint. For error classification, the controlled benchmark reported in this report uses a 55-case DeepSeek v3.2 run with and without execution feedback. Additional saved output sheets were reviewed qualitatively to compare prompt behaviour across other model outputs archived in the workspace.

The **second task** involved conducting a literature review and writing a synthesis paper on the trade-offs between training code-focused LLMs on a single programming language versus multiple programming languages. The paper brings together evidence on model scale, cross-lingual transfer, code repair, and deployment constraints, and organizes that material into a decision framework based on model scale, task abstraction, and language similarity.

The **third task** was the design and implementation of a full-stack adaptive question sequencing system for the Yaksh coding education platform. The system is built around a deterministic state machine that governs question selection across six concept levels (Conditionals, Basic Loops, Loops with Lists, Nested Loops, Strings, and Functions), cycles through error-type priorities, enforces mastery through a minimum of eight attempts and a streak of four correct answers, uses forced resolve mode after incorrect submissions, and switches to seeded random mode when a student remains stuck after the attempt threshold. The backend was implemented with FastAPI and SQLAlchemy async sessions, while a lightweight HTML/CSS/JavaScript frontend provides a demo interface. A separate proposal document records the database schema, API contracts, Yaksh integration plan, and deployment considerations.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Programming platforms often return raw runtime messages and fixed question sequences that are hard for beginners to use effectively. This internship focused on three pieces needed to improve that loop in Yaksh [5] identifying what went wrong in a student submission, generating a useful hint without giving away the answer, and choosing what question should come next.

In traditional coding platforms, when students encounter errors they receive generic error messages that do not guide them toward understanding the root cause. A pedagogically effective tutoring system must:

1. **Detect and classify errors** from student code and execution feedback accurately.
2. **Generate hints** that guide students to discover solutions themselves, rather than providing direct answers.
3. **Sequence questions adaptively** based on student performance to ensure mastery of concepts before progression.

Each requirement presents distinct technical challenges, from prompt engineering and NLP-based analysis for LLMs to designing deterministic state machines for adaptive learning.

1.2 Objectives

The primary objectives of this internship were:

- Develop and evaluate an LLM-based error-classification workflow on a labeled set of student submissions, and inspect prompt behaviour on additional saved model outputs.
- Design a multi-stage prompt pipeline that improves hint quality through structured error analysis and pedagogical strategy selection.
- Conduct a literature review on single-language versus multi-language training strategies for code-focused LLMs.
- Implement a full-stack adaptive question sequencing system with mastery-based progression across six concept levels (Conditionals, Basic Loops, Loops with Lists, Nested Loops, Strings, and Functions) for the Yaksh platform.

1.3 Scope and Contributions

This work spans three interconnected tasks that together address the full feedback loop in coding education: detecting *what* is wrong, determining *how* to communicate the error, and deciding *when* to present the next challenge. Table 1 summarises the scope of each task.

Table 1: Summary of internship tasks and deliverables.

Task	Description	Key Deliverables
1	Error detection, classification, and multi-stage hint generation	Analyzer, pipeline, prompts
2	Literature review on code LLM training strategies	Modular LaTeX paper
3	Adaptive question sequencing system for Yaksh	Full-stack prototype

1.4 Report Organization

Section 2 describes the error detection and multi-stage analysis work. Section 3 covers the literature review paper. Section 4 details the adaptive question sequencing system. Section 5 presents conclusions and future directions.

2 Error Detection and Hint Generation

This task involved three connected sub-tasks: building an improved response analyzer for evaluating LLM-generated hints, developing error-classification prompts and benchmarking one controlled classifier run, and designing a multi-stage hint generation pipeline. Together, these components support both the analysis and the generation of tutoring hints for Python programming students.

2.1 Response Analyzer

The response analyzer is a Python-based tool that evaluates LLM-generated educational responses across six analytical dimensions, summarised in Table 2. Each dimension measures a distinct aspect of response quality: code validation uses Python's AST parser to verify code completeness; pedagogical analysis counts guiding words, Socratic questions, and metacognitive prompts; educational tone classifies the response as supportive, neutral, critical, or discouraging; error relevance scores how well the response addresses the specific error; readability metrics estimate the grade level needed to understand the response; and contextual analysis checks whether the response references the student's code and execution feedback.

Table 2: Analytical dimensions of the response analyzer.

Dimension	**Techniques Used**	**Key Outputs**
Code Validation	Python AST parsing, regex extraction	Completeness score (0–3)
Pedagogical Analysis	Word-boundary regex, pattern matching	Socratic, metacognitive counts
Educational Tone	TextBlob sentiment + domain rules	Supportive / neutral / critical / discouraging
Error Relevance	Keyword matching against 9 error types	Relevance score (0–1)
Readability	Flesch, Gunning Fog, SMOG indices	Grade-level estimates
Contextual Analysis	Stopword-filtered word overlap	Code/feedback reference flags

At the core of the analyzer is a code validation module that extracts code snippets from LLM responses and validates them using Python's AST (Abstract Syntax Tree) module, eliminating false positives from natural language text that merely resembles code. Each extracted snippet receives a completeness score on a scale of 0 to 3, classifying it as keywords-only, a partial snippet, a complete function, or a complete function with control flow logic.

The analyzer computes a range of pedagogical metrics. It detects guiding language (words such as "try," "consider," "explore") using word-boundary-aware regex patterns that prevent false matches—for instance, "try" embedded within "country" is correctly ignored. Socratic questioning patterns are identified through regex rules matching constructs like

"what would happen if" and "how does this work." Metacognitive prompts such as "explain your thinking" and "walk me through your approach" are counted separately, as are direct solution indicators ("here is the answer," "change it to") and growth mindset language ("learn," "practice," "improve").

The educational tone classification module combines TextBlob sentiment polarity and subjectivity scores with domain-specific indicators to label each response as *supportive*, *neutral*, *critical*, or *discouraging*. An error-specific relevance scoring system maps nine Python exception types (AttributeError, TypeError, NameError, IndexError, ValueError, SyntaxError, ZeroDivisionError, KeyError, RuntimeError) to their associated keyword lists and calculates a normalized relevance score reflecting how well the response addresses the specific error.

Readability is assessed through Flesch Reading Ease, Gunning Fog Index, and SMOG Index calculations. A contextual analysis module evaluates whether the response references the student's code and execution feedback, measures question–response word overlap after stopword removal, and flags responses that are too short or appear generic. For performance, the analyzer caches spaCy (an open-source Python library for natural language processing) document objects to avoid redundant NLP pipeline invocations when processing large batches.

The main processing function reads an Excel file with flexible column name matching, computes all metrics for each row, and outputs an enriched Excel file with two manual evaluation columns (Acceptable and Issues) appended for human review. A test suite validates the analyzer across five dimensions: code detection accuracy, pedagogical metric sensitivity, error relevance scoring, pattern matching false-positive prevention, and readability metric correctness.

2.2 Prompt Design and Evaluation

We designed two structured prompt templates for interacting with LLMs in an educational context:

1. **Tutor Prompt:** Instructs the LLM to act as a Python tutor, enforcing strict rules: never write or correct code directly, use only Python, and provide hints or questions rather than solutions. It accepts the student's question, code, and execution feedback as inputs and generates a single focused hint of two to three sentences.
2. **Classifier Prompt:** Instructs the LLM to classify the error type from a fixed taxonomy of seven categories (AttributeError, IndexError, NameError, OtherRuntimeError, TypeError, ValueError, ZeroDivisionError). The classifier is constrained to respond with only the error name, enabling automated downstream processing.

A comparison analysis tool was developed that loads AI-classified error types alongside original ground-truth labels, computes overall accuracy and per-error-type accuracy, identifies mismatches, and exports mismatched cases for manual review. The dataset comprised student code submissions with verified error labels across multiple error categories. The controlled comparison reported here uses the saved DeepSeek v3.2 runs because complete paired outputs were available both with and without execution feedback. Other archived sheets preserve outputs from additional model runs, but those were used for qualitative comparison rather than for the paired 55-case benchmark. The

mismatched cases were audited to identify which error types the model confused most frequently, and this analysis fed back into prompt refinements.

2.2.1 Classifier Evaluation Results

The controlled comparison was carried out on a benchmark of 55 labeled student submissions spanning seven runtime-error classes. When the classifier received only the problem statement and the student code, it matched the ground-truth label on 35 of 55 cases, corresponding to an overall accuracy of 63.64%. When execution feedback was added, accuracy increased to 51 of 55 cases, or 92.73%. The gain was not uniform across error types. It was most pronounced for categories where the same code could plausibly suggest multiple failure modes unless the actual runtime message was visible, particularly TypeError, AttributeError, ValueError, and the broad OtherRuntimeError bucket.

Table 3: Error-classification accuracy with and without execution feedback on 55 labeled cases.

Error Type	**Without Feedback**	**With Feedback**
Overall	35/55 (63.64%)	51/55 (92.73%)
OtherRuntimeError	2/10 (20.0%)	6/10 (60.0%)
ValueError	5/8 (62.5%)	8/8 (100.0%)
TypeError	1/8 (12.5%)	8/8 (100.0%)
ZeroDivisionError	8/8 (100.0%)	8/8 (100.0%)
AttributeError	5/7 (71.4%)	7/7 (100.0%)
IndexError	7/7 (100.0%)	7/7 (100.0%)
NameError	7/7 (100.0%)	7/7 (100.0%)

The mismatch analysis clarified where the model still struggled. In one quadratic-equation submission, the no-feedback classifier predicted NameError because the code looked structurally fragile, while the platform label was OtherRuntimeError because the actual failure was a wrong-answer boundary case rather than a crashed execution. In another case involving dictionary membership, the true runtime message was AttributeError ("str" object has no attribute "items"), but the no-feedback classifier predicted TypeError by reasoning from the likely operation rather than from the observed object method failure. A third failure mode appeared in a quadratic-solver submission whose function signature did not match the test harness: the true label was TypeError, but the no-feedback classifier predicted ZeroDivisionError from the mathematics of the problem statement instead of from the call-site mismatch. Even after execution feedback was added, the remaining errors were concentrated in OtherRuntimeError cases where platform-level labels merged distinct underlying exceptions such as UnboundLocalError and EOFError into a single catch-all category.

2.3 Multi-Stage Hint Generation Pipeline

Single-stage hint generation, where an LLM directly produces a hint from student data, suffers from inconsistent quality, lack of pedagogical control, and difficulty in debugging poor outputs. To address these limitations we designed a three-stage pipeline that sep-

arates error analysis, pedagogical strategy selection, and hint generation into distinct stages, each operating at a different temperature to balance determinism with creativity.

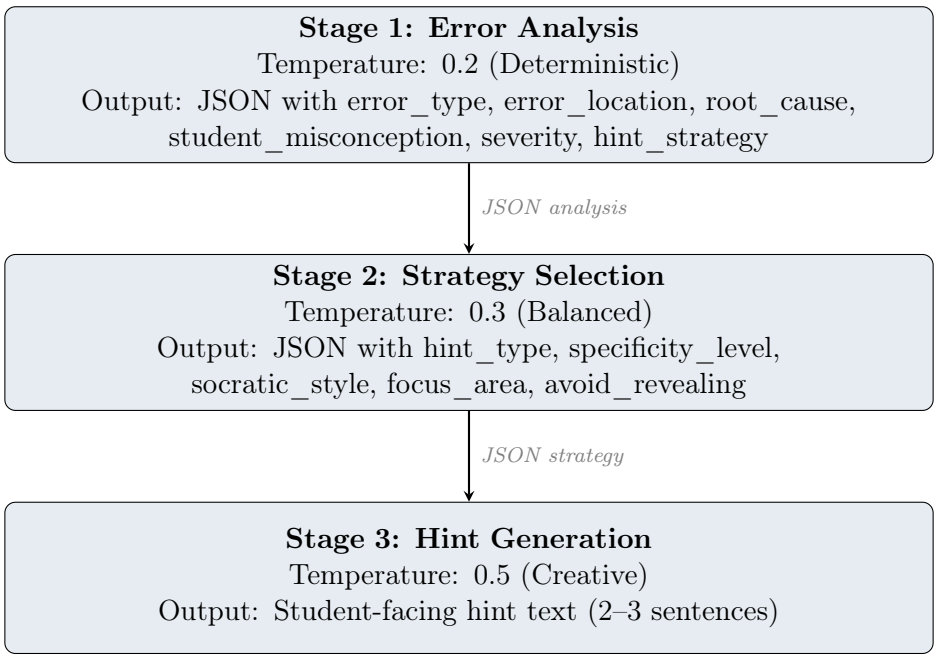

Figure 1: Three-stage hint generation pipeline architecture.

Stage 1 performs systematic error analysis at temperature 0.2 to ensure deterministic, consistent classification. It takes the student's question, code, and execution feedback as input and produces a structured JSON containing: the error type (classified as syntax, runtime, logic, algorithm, or test_case error), the error location, a technical root cause explanation, the likely student misconception, the severity (blocking or non-blocking), and a recommended hint strategy.

Stage 2 consumes the Stage 1 JSON and selects a pedagogical approach at temperature 0.3. It outputs a strategy JSON specifying the hint type, specificity level (vague, medium, or specific), the Socratic style, the focus area, and an avoid-revealing list that explicitly names concepts or code patterns the hint must not mention. Table 4 shows the strategy rules encoded at this stage.

Table 4: Pedagogical strategy rules encoded in Stage 2.

Error Type	Hint Type	Specificity
Syntax	Syntactic hint	Medium
Logic	Conceptual question	Vague
Algorithm	Algorithmic guidance	Medium
Test-case	Conceptual question	Medium

Stage 3 generates the actual student-facing hint at temperature 0.5, allowing natural language variation while remaining within the guardrails established by Stage 2. It receives both the analysis and strategy JSONs and produces a plain-text hint of two to

three sentences that follows the specified format constraints: no code blocks, no direct solutions, and adherence to the chosen hint type and Socratic style.

The pipeline was implemented in Python using the OpenRouter API with the DeepSeek v3.2 model. Practical considerations included rate limiting (0.5 s between stages, 1.5 s between test cases), robust JSON parsing with markdown fence stripping, token usage tracking at each stage, and error handling that preserves partial results when a stage fails. Table 5 summarises the five archived test cases. In that sample, Stage 1 totals ranged from 632 to 994 tokens, Stage 2 from 656 to 675 tokens, and Stage 3 from 397 to 423 tokens.

Table 5: Multi-stage pipeline results across five test cases.

Metric	**Value**
Completed runs	5/5
Average total tokens per case	1,805
Stage 1 tokens (avg.)	733 (40.6%)
Stage 2 tokens (avg.)	663 (36.7%)
Stage 3 tokens (avg.)	409 (22.7%)
JSON parsing success rate	100%

Manual inspection of the five archived outputs showed that the generated hints stayed within the intended format constraints: they were short, did not include code blocks, and did not provide direct solutions. The multi-stage approach is still more token-heavy than a rough single-stage baseline (about 1,805 versus about 1,000 tokens per hint in this sample), but the extra structure makes the system easier to inspect and debug because each stage can be reviewed separately.

2.3.1 Worked Multi-Stage Example

One representative case illustrates how the pipeline translates a concrete student failure into a controlled pedagogical response. The task asked for the two solutions of a quadratic equation. For the boundary case $(a, b, c) = (0, 0, 1)$, the expected output was an empty list, but the student submission returned "No solution." The relevant part of the submission is shown below.

Listing 1: Student code fragment from the worked multi-stage example.

```
def main(a, b, c):
    if a == 0:
        if b == 0:
            if c == 0:
                return "Infinite solutions"
            return "No solution"
```

Stage 1 treated this as a logic error rather than a syntax or runtime failure, because the code executed successfully but produced the wrong output format for the linear-equation branch.

Listing 2: Stage 1 analysis JSON for the worked example.

```
{
  "error_type": "logic_error",
  "error_location": "Handling of linear equation case (a
    =0)",
  "root_cause": "The code returns a tuple format for
    linear equations that doesn't match the expected
    output format for quadratic equations",
  "student_misconception": "Not maintaining consistent
    output format between linear and quadratic cases",
  "severity": "non_blocking",
  "hint_strategy": "algorithm_guidance"
}
```

Stage 2 then converted this diagnosis into an instructional plan. Instead of naming the exact output format, it selected an algorithmic hint with medium specificity and explicitly recorded which details should not be revealed.

Listing 3: Stage 2 pedagogical strategy JSON for the worked example.

```
{
  "hint_type": "algorithmic_guidance",
  "specificity_level": "medium",
  "socratic_style": "hint_then_question",
  "focus_area": "algorithm_logic",
  "avoid_revealing": [
    "exact output format",
    "tuple structure"
  ]
}
```

Finally, Stage 3 generated the student-facing hint:

Listing 4: Final Stage 3 hint for the worked example.

```
Consider how the output for linear equations aligns with
   the structure expected for quadratic equations. How can
    you ensure consistency in the way results are
   presented across both cases?
```

This example captures the main advantage of the multi-stage design. The system first identifies the failure as an output-consistency problem, then constrains the pedagogical strategy, and only at the end produces the natural-language hint. Because each stage is inspectable, it is possible to see whether a weak final hint came from a poor diagnosis, a poor strategy choice, or a poor wording decision.

3 Literature Review on Code LLM Training Strategies

The second task involved conducting a literature review and writing a synthesis paper on the trade-offs between training code-focused LLMs on a single programming language versus multiple programming languages.

3.1 Research Questions

The central research question—*when is single-language training better, and when is multi-language training better?*—was decomposed into sub-questions examining:

- Model scale effects on the monolingual vs. multilingual trade-off
- Cross-lingual transfer evidence and zero-shot translation capabilities
- Specialisation advantages of single-language training
- Impact of training strategies on code repair and inference efficiency

3.2 Methodology

The review followed a focused evidence-synthesis workflow rather than a broad narrative survey. Papers were sourced from arXiv and major machine-learning and NLP venues, then filtered using three explicit criteria: the study had to compare single-language and multi-language training directly, use execution-based or task-grounded evaluation, and analyse models at or above roughly the 100M-parameter scale. Within the project workspace, four primary comparative studies met these criteria and became the core technical evidence base. These covered multilingual scaling behaviour, cross-lingual transfer, data-composition ablations, and multilingual ensemble inference. The final synthesis was then expanded with five industrial case studies drawn from major production model families in order to connect controlled empirical findings with real deployment practice.

The review workflow was document-driven. For each paper, source text and key figures were extracted into the analysis workspace, then annotated by training strategy, model scale, task abstraction, benchmark type, and deployment relevance. This made it possible to separate questions about pre-training scale, fine-tuning strategy, cross-lingual transfer, and practical serving constraints instead of mixing them under a single "multilingual versus monolingual" label. Studies that did not provide a direct comparison, did not report task-grounded evaluation, or focused on unrelated optimisation problems were used only as background material and not as core evidence in the decision framework.

3.3 Paper Structure and Outputs

The resulting paper was developed as a standalone LaTeX project backed by a separate analysis workspace. The compiled manuscript contained nine figures, six tables, and a bibliography of ten references. Supporting assets included four Python figure-generation scripts, six TikZ figure-source files, eight external figure files, extracted text from the four primary comparative papers, and source-material folders for the broader case-study analysis. This separation between the analysis workspace and the final paper package made the drafting process easier to audit and revise.

Table 6: Task 2 literature-review workflow and artifact statistics.

Item	**Count / Description**
Primary comparative studies reviewed in detail	4
Industrial case studies added in final synthesis	5
Compiled figures in final paper	9
Compiled tables in final paper	6
Bibliography entries in standalone paper	10
Python figure-generation scripts	4
TikZ figure-source files	6
External figure assets in paper package	8

The toolchain combined manual reading with lightweight automation. LaTeX and BibTeX were used for the paper itself, while Python scripts were written to regenerate comparison charts for scaling trends, zero-shot translation behaviour, code-repair visualisations, and benchmark-specific summaries. The output of this task was a paper package with data extracts, plotting scripts, figure assets, and revision notes that documented how the argument evolved.

3.4 Key Contributions

One outcome of the survey was a decision framework organized around three variables: model scale, task abstraction, and language similarity. The paper also distinguished between *pre-training* and *fine-tuning*, because several sources used "monolingual" and "multilingual" to describe different stages of training rather than the same experimental setting. Python scripts were developed to generate charts and figures from extracted data.

The synthesis suggested a scale-dependent pattern rather than a single universal rule. In the studies reviewed here, smaller decoder-only models below roughly one billion parameters more often benefited from single-language focus, while larger models above roughly two billion parameters more often benefited from multilingual transfer [1, 3, 4]. The same review also suggested that syntactically sensitive tasks, such as exact code repair or narrow specialization, tend to benefit from language-specific tuning, whereas broader semantic or cross-lingual tasks benefit more from multilingual training or multilingual foundations followed by later specialization [2].

The review also summarized the deployment side of these model choices. Several case studies in the paper point to a practical jump around the 34B scale, where serving often moves from a single high-memory card to multi-GPU infrastructure. The DeepSeek case study also illustrates the non-linear relationship between accuracy and compute: moving from 6.7B to 33B improves HumanEval performance, but it also raises memory and serving costs substantially [4]. In a different setting, the Begolli study showed that a much smaller domain-specific fine-tuning stage can still deliver a useful quality gain for a specialized industrial task without repeating foundation-model pre-training [2].

4 Adaptive Question Sequencing

The third task involved designing and implementing an adaptive question sequencing system for the Yaksh coding education platform [5]. The proposed system selects questions on the basis of student performance, enforces mastery before progression, and targets different error types within each concept.

4.1 Concept Progression and Error Types

The system organises content into six concepts arranged in a fixed linear progression, as shown in Table 7. The progression moves from Conditionals through Basic Loops, Loops with Lists, Nested Loops, and Strings, culminating in Functions. Each concept introduces more complex programming constructs and demands deeper understanding. Students must master each concept sequentially without skipping, ensuring a solid foundation before advancing.

Table 7: Concept progression chain and associated error types.

ID	Concept	Error-Type Priority Order
C1	Conditionals (If-else)	Syntax, Conceptual, Input, Variable, ArrayString, Condition, Loop, Computation, Branching, Output
C2	Basic Loops (For/While)	
C3	Loops with Lists	
C4	Nested Loops	
C5	Strings	
C6	Functions	Syntax, Conceptual, **Function**, Input, Variable, ArrayString, Condition, Loop, Computation, Branching, Output

Each concept is subdivided into error types (10 for most concepts, 11 for Functions). The system cycles through these types deterministically during normal mode.

4.2 Architecture

The system follows a modular architecture with clear separation of concerns, as illustrated in Figure 2. A FastAPI application (FastAPI, a modern Python web framework for building APIs) serves as the entry point, registering API routes and mounting a lightweight demo frontend. Middleware layers handle CORS for development, request logging via a request logging middleware, and state validation via a stateless API middleware. The core sequencing engine implements the deterministic algorithm, a question service layer orchestrates state management and engine invocations, and a storage service backed by SQLAlchemy (SQLAlchemy, a Python SQL toolkit and ORM) with async sessions manages all database operations.

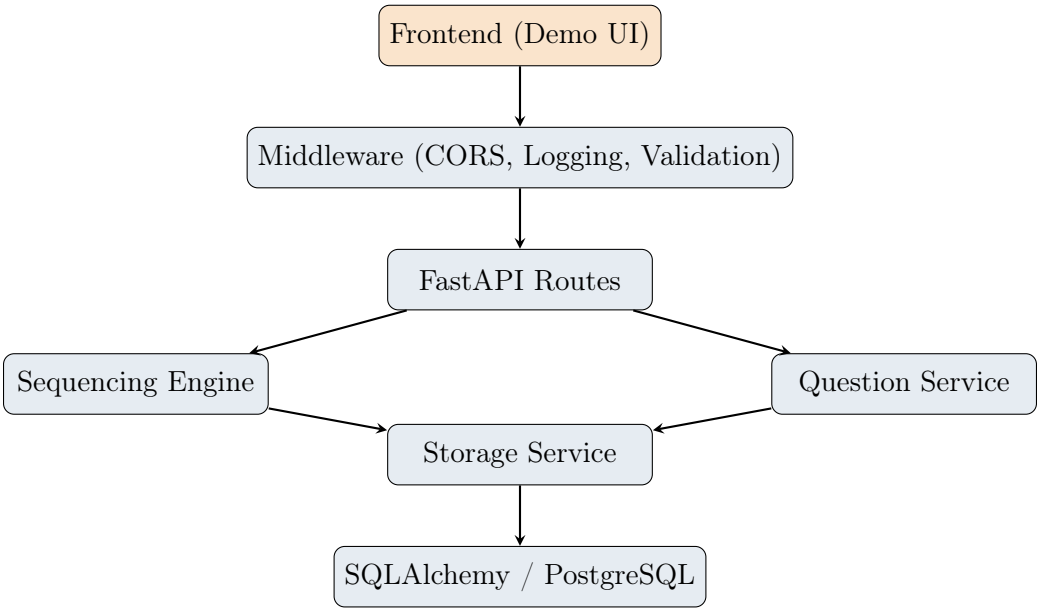

Figure 2: Adaptive sequencing module architecture.

4.3 Sequencing Algorithm

The core of the system is a deterministic state machine governed by two key thresholds: eight minimum unique-question attempts before mastery eligibility, and a mastery streak of four consecutive correct answers on new questions without hint use. Algorithm 1 formalises this logic. The notation **Input** denotes the inputs to the algorithm (student state S and question bank Q), and **Output** denotes its return value (the next question q).

Algorithm 1: Adaptive Question Sequencing

Input: Student state S, Question bank Q
Output: Next question q

if $S.attempts \geq 8$ ***and*** $S.streak \geq 4$ **then**
    Mark concept mastered;
    Unlock and advance to next concept;
    Reset all state variables;
    return next question from new concept;
if $S.attempts \geq 8$ ***and*** $S.streak < 4$ **then**
    S.random_mode $\leftarrow$ `true`;
    S.forced_resolve $\leftarrow$ `false`;
if $S.random\_mode$ **then**
    $e \leftarrow$ random error type via SHA-256 seed;
else
    $e \leftarrow$ next sequential error type from priority order;
$q \leftarrow$ RETRIEVE(S.concept, e, S.recent_ids);
Update bookkeeping (recent IDs, attempts, presented set);
return q;

The mastery rule requires both conditions to be satisfied simultaneously: at least eight

attempts in the concept and a current streak of four or more correct answers on new questions without hint use. When mastery is achieved, the current concept is marked as mastered, the next concept in the progression is unlocked, and all state variables (attempts, streak, error type index, mode flags) are reset.

Forced resolve mode activates when a student answers incorrectly or uses a hint (except in random mode). In this mode, the same question is served again for retry, but retries do not count as new attempts toward the mastery threshold. The streak is reset to zero, and the mode persists until the student answers correctly.

Random mode activates when a student accumulates eight or more attempts without achieving the required streak of four. Error types are selected using a deterministic seed derived via SHA-256 from the student ID, current concept, attempt count, and last question ID, ensuring reproducibility while preventing predictable cycling. Forced resolve is deactivated in random mode.

Question retrieval follows a four-step fallback: (1) exact match on concept and error type excluding recent questions, (2) reset exclusion and retry, (3) any question in the concept, (4) raise an error indicating the question bank is exhausted for this concept. Table 8 summarises the state transitions that govern the sequencing engine.

Table 8: State transition rules for the sequencing engine.

From State	Trigger	To State
Normal	Incorrect answer or hint used	Forced Resolve
Forced Resolve	Correct answer (no hint)	Normal
Normal	Attempts ≥ 8, Streak < 4	Random Mode
Any	Attempts ≥ 8, Streak ≥ 4	Mastery $\rightarrow$ Next Concept

4.4 Backend Implementation

The backend exposes three RESTful endpoints through the FastAPI framework:

- `POST /next-question` — retrieves the next question based on the student's current state.
- `POST /submit-answer` — records an answer submission, updates state, checks for mastery, and returns the next question along with feedback.
- `POST /reset-concept` — resets a student's state for a specific concept (administrative/testing).

All endpoints accept and return JSON with full state snapshots for frontend synchronisation. A health-check endpoint reports service status and loaded question count.

The data layer uses SQLAlchemy async sessions with two core models: one for storing question data (question ID, concept, error type, difficulty, and content) and another for tracking per-student state (current concept, error type index, attempts, streak, forced resolve and random mode flags, recent and presented question ID lists, and mastered and unlocked concept lists). An interaction log records all state transitions for audit and analytics. The database is seeded from a JSON question bank on first startup through a FastAPI lifespan event handler. The system runs on PostgreSQL, with SQLAlchemy managing the schema and async sessions handling concurrent requests.

The sequencing engine implements the core algorithm as an async function with clearly separated internal helpers: a function that activates random mode when needed, a function that chooses the target error type, a function that retrieves questions through a four-step fallback, and a post-selection bookkeeping function. A separate submission processor handles answer evaluation, streak management, forced resolve logic, and mastery detection.

4.5 Frontend and Documentation

A lightweight demo UI built with HTML, CSS, and vanilla JavaScript provides a question display with concept and error type indicators, an answer submission interface with correctness feedback, streak and attempt counters, and visual indicators for forced resolve and random modes. The frontend executes student code in-browser via Pyodide, a port of CPython compiled to WebAssembly, enabling browser-side Python execution without a remote kernel. The UI communicates with the backend through the three RESTful endpoints and is served as static files by FastAPI at the /app route. The system includes tests covering the sequencing engine state machine, API endpoint integration, demo payload mounting, seed question parsing, and stateful API workflows. A detailed proposal document was authored covering the problem statement, PostgreSQL database schema design with indexing strategy, the complete algorithm specification with worked examples, API contracts with full request and response schemas, an integration strategy with the Yaksh platform, deployment and DevOps considerations including Docker Compose for local development, horizontal scaling and caching, security and access control design, and a staged rollout plan with monitoring and rollback procedures.

4.6 Current Question-Bank Coverage

Although the sequencing algorithm and API support all six concepts in the progression chain, the currently seeded question bank is uneven. The prototype JSON bank contains 194 questions in total, but most of them are concentrated in C1, C3, and C5. This means that the engine logic is already complete enough to exercise mastery, forced resolve, random mode, and fallback retrieval across the full concept chain, while content coverage is still sparse for several later concepts.

Table 9: Current seeded question-bank coverage by concept and error type.

Concept	Questions	Represented Error Types	Coverage Note
C1	60	Syntax(5), Conceptual(5), Input(5), Variable(5), Condition(18), Computation(6), Branching(11), Output(5)	Strong seed; missing ArrayString and Loop
C2	7	Syntax(2), Conceptual(2), Input(1), Loop(2)	Sparse seed
C3	62	Syntax(5), Conceptual(5), Input(5), Variable(5), ArrayString(10), Condition(6), Loop(11), Computation(5), Branching(5), Output(5)	Full 10/10 coverage
C4	4	Syntax(1), Conceptual(1), Loop(1), Computation(1)	Sparse seed
C5	57	Syntax(5), Conceptual(5), Input(5), Variable(5), ArrayString(11), Condition(6), Loop(5), Computation(5), Branching(5), Output(5)	Full 10/10 coverage
C6	4	Syntax(1), Conceptual(1), Function(2)	Early seed only

This distribution has two practical consequences. First, the prototype already supports realistic end-to-end behaviour for well-seeded concepts, especially C1, C3, and C5, where the engine can cycle through most or all priority classes without immediately falling back to concept-level retrieval. Second, later concepts still need content expansion before the system can be treated as instructionally balanced. In other words, the sequencing algorithm is already in place, but the content bank has not yet caught up with the full breadth of the design.

5 Conclusion

5.1 Summary of Contributions

This internship contributed to three parts of an AI-assisted tutoring workflow:

1. **Error Analysis and Hint Generation** — We built a six-dimensional response analyzer for evaluating LLM tutoring quality and a three-stage pipeline for generating non-revealing hints. The pipeline separates error analysis (T=0.2), strategy selection (T=0.3), and hint generation (T=0.5), which makes it easier to inspect each stage separately. On a 55-case classifier benchmark, adding execution feedback improved accuracy from 63.64% to 92.73%. In the archived five-case pipeline sample, all runs completed without API or JSON-parsing failure.
2. **Literature Synthesis** — We reviewed and synthesised evidence on single-language versus multi-language training strategies for code-focused LLMs and organized the results around model scale, task abstraction, and language similarity. The resulting standalone paper package contained nine figures, six tables, a ten-entry bibliography, and a supporting set of Python figure-generation scripts and extracted source materials.
3. **Adaptive Question Sequencing** — We designed and implemented a deterministic state machine that governs personalised learning paths, cycling through error types across six concept levels with forced resolve and random modes. In random mode, error types are selected using a deterministic seed derived via SHA-256 to ensure reproducibility while preventing predictable cycling. The full-stack prototype uses FastAPI, SQLAlchemy, and PostgreSQL with a browser-based Python execution environment via Pyodide. The current seeded question bank contains 194 questions, with strong coverage in C1, C3, and C5 and lighter seeding in C2, C4, and C6, so the algorithm is in place even though content expansion is still needed before Yaksh integration.

These three components address the full feedback loop in coding education: detecting what is wrong, determining how to communicate it, and deciding when to present the next challenge.

5.2 Limitations and Future Work

This internship report documents working prototypes and focused evaluations rather than full classroom deployment studies. The classifier benchmark in Task 1 contains 55 labeled submissions, which is sufficient to reveal clear engineering trends but still too small to support broad claims about robustness across the full range of beginner Python mistakes. Likewise, the multi-stage hint pipeline was validated on five representative cases. Those results are useful for understanding traceability, token cost, and prompt behaviour, but they do not yet replace a larger-scale evaluation with many more student submissions and a stronger baseline comparison against single-stage prompting or non-LLM tutoring heuristics.

The adaptive sequencing system also remains a prototype in an important sense: the engine, API layer, and state transitions are implemented, but the seeded question bank is uneven across concepts. C1, C3, and C5 are well represented, while C2, C4, and C6 still need broader error-type coverage before the content can support stable long-run

progression in production. In addition, the frontend and backend have been validated through tests and local workflows rather than through a live Yaksh deployment or a classroom study. No controlled user study has yet been carried out to measure learning gains, frustration reduction, hint usefulness, or progression quality under real student behaviour.

The next steps are straightforward. Task 1 should be extended with a larger labeled benchmark, stronger human evaluation of hint quality, and side-by-side comparison against simpler tutoring baselines. Task 3 should be expanded by filling the sparse concepts in the question bank, instrumenting the system for analytics, and integrating the module into Yaksh so that state transitions can be observed under real usage. At the study level, the combined system should eventually be evaluated in a classroom or controlled lab setting to test whether better error diagnosis, better hints, and adaptive sequencing translate into measurable gains in student learning rather than just cleaner system behaviour.

References

References

[1] B. Athiwaratkun et al., "Multi-lingual Evaluation of Code Generation Models," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[2] I. Begolli, M. Aksoy, and D. Neider, "Fine-Tuning Multilingual Language Models for Code Review: An Empirical Study on Industrial C# Projects," in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025.

[3] B. Rozière et al., "Code Llama: Open Foundation Models for Code," *arXiv preprint arXiv:2308.12950*, 2023.

[4] D. Guo et al., "DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[5] Yaksh, "Live Python coding environment." https://yaksh.fossee.in/