Internship Completion Report

Submitted for the Award of Internship Completion Certificate

Adaptive Question Sequencing, Error Categorization, and Literature Synthesis

Sarthak Prasad Kulkarni

Bachelor of Technology (B.Tech)
Information Technology, Semester 6 (Third Year)
Vidyalankar Institute of Technology
Expected Graduation: 2027

Email: sarthakkulkarni12@outlook.com, icsarthak9@gmail.com
LinkedIn: LinkedIn Profile
GitHub: GitHub Profile

FOSSEE Python Research Intern (AI/LLM Research)

Internship Period: October 17, 2025 – April 15, 2026

Supervisors

Prof. Dr. Kushal Shah
Prof. Prabhu Ramachandran

April 22, 2026

FOSSEE Project
IIT Bombay

Contents

Executive Summary 3

Abstract 4

Acknowledgements 5

1 Introduction and Objectives 6
- 1.1 Background and Context of the Internship . . . 6
- 1.2 Core Objectives and Scope of Work . . . 6
- 1.3 Chronological Narrative and Milestones . . . 6

2 Phase 1: Data Analytics and Error Categorization (October 2025 – November 2025) 10
- 2.1 Core Objective and Tool Selection . . . 10
- 2.2 Initial Behavioral Analysis and Error Resolution Rates . . . 10
- 2.3 Refined Error Categorization via K-Means Clustering . . . 10
- 2.4 Hierarchical Sub-Categorization of Type Errors . . . 11
- 2.5 Dataset Lineage, Data Cleaning, and Deduplication . . . 11

3 Phase 2: Open-Source LLM Evaluation and Prompting (October 2025 – November 2025) 12
- 3.1 Comprehensive Project Execution and Evaluation . . . 12
  - 3.1.1 Core Objective and Model Selection . . . 12
  - 3.1.2 Dataset Shortlisting and Evaluation Infrastructure . . . 12
  - 3.1.3 AI-Enriched Error Dataset Generation Pipeline . . . 12
  - 3.1.4 Prompt Engineering and the "Thin Line" Constraint . . . 13
  - 3.1.5 Independent Evaluation of DeepSeek v3.2 Express . . . 13
  - 3.1.6 Multistage Prompting and Architectural Pivot . . . 13
  - 3.1.7 Automated NLP and Linguistic Scoring . . . 14

4 Phase 3: Literature Review on Code LLM Training (December 2025 – January 2026) 15
- 4.1 Scope and Core Research Objective . . . 15
- 4.2 Research Methodology and Comparative Analysis . . . 15
- 4.3 Original Data Extraction and Visualization . . . 15
- 4.4 Document Expansion and LaTeX Modularization . . . 16
- 4.5 Mid-Scale Model Analysis (30B–70B Parameters) . . . 16
- 4.6 Final Formatting and Academic Integration . . . 16
- 4.7 Key Theoretical Findings and Decision Framework . . . 17
- 4.8 Document Architecture and Team Coordination . . . 18

5 Phase 4: Adaptive Question Sequencing Algorithm (February 2026 – April 2026) 19
- 5.1 Problem Statement and Architectural Pivot . . . 19
- 5.2 Core Algorithmic Design . . . 19
  - 5.2.1 Linear Concept Progression . . . 19
  - 5.2.2 Intra-Concept Sequencing and Error Categories . . . 19
  - 5.2.3 Mastery Gates . . . 19

5.2.4 Strict Evaluation and Forced Re-Solves 19
5.2.5 System States: Sequential vs. Random Mode 20
5.3 Question Pool Engineering and Curriculum Design 20
5.4 Technical Architecture and Implementation 20
5.4.1 Stateless API Design 20
5.4.2 PostgreSQL and Schema Management 21
5.4.3 Deployment Efficiency and Scalability 21
5.4.4 Backend Integration and Debugging 21
5.5 Development Milestones and Documentation 21
5.6 Project Review and Final Deployment 22

6 Technical Contributions 23
6.1 Backend: Sequencing Engine and Stateless API 23
6.2 Error Categorization and Tutor Prompting 23
6.3 Frontend Demo and Integration 23
6.4 Data, Evaluation, and Reproducibility 25

7 Cross-Phase Insights and Design Lessons 26
7.1 Research-Grounded Design Rationale 26
7.2 Implementation Challenges and Lessons Learned 26

8 Conclusion 27
8.1 Summary of the Internship Journey 27
8.2 Impact on the YAKSH Ecosystem 27
8.3 Academic Fulfillment and Final Handoff 28
8.4 Next Steps and Future Work 28

References 29

Codebase Links 30

EXECUTIVE SUMMARY

During the seven-month FOSSEE Python Research Internship, I completed a research-and-engineering portfolio aimed at improving AI-assisted programming education for beginners and strengthening the YAKSH ecosystem. The work combined large-scale analysis of student submission data using Python and `pandas`, structured evaluation of open-source tutoring-oriented LLM workflows, formal literature synthesis in IEEE style, and full-stack system prototyping with FastAPI, stateless API contracts, a Vite/React demonstration interface, Firestore-backed demo persistence, and PostgreSQL-oriented production integration planning. The principal deliverables were a taxonomy of novice Python errors, a reproducible evaluation framework for tutoring models, a literature review on single-language versus multi-language code-model training, and a deployable adaptive question sequencing prototype for YAKSH. Together, these outcomes demonstrate measurable research impact, practical software delivery, and a clear contribution toward more explainable, scalable, and learner-centered programming support.

ABSTRACT

This report documents the research, evaluation, and engineering outcomes of a seven-month FOSSEE Python Research Internship (October 2025 – April 2026) at IIT Bombay, focused on advancing AI-assisted programming education for novice learners. The internship addressed three persistent challenges in educational coding platforms: high failure rates driven by recurring beginner errors, reliance on proprietary or opaque AI tutoring models, and static question delivery that impedes conceptual mastery.

The work progressed through four integrated phases. First, large-scale analysis of 14,408 student interactions from the PyPal platform was conducted using Python and `pandas`. Ambiguous compiler logs were cleaned, numerically encoded, and clustered via K-means, yielding a structured error taxonomy that revealed Function Argument Mismatches as the dominant failure mode, accounting for 73.9% of all unsuccessful attempts. Second, open-source code LLMs were evaluated for pedagogical tutoring, with DeepSeek v3.2 Express benchmarked using a reproducible rubric, LLM-as-a-judge scoring, and constrained prompt engineering. The model achieved 100% fix correctness and perfect pedagogical safety ratings, while a composite debugging quality score of 0.686 highlighted opportunities for improved root-cause diagnostic clarity in edge cases. Third, a systematic literature review synthesized findings from 20 reviewed studies on single- versus multi-language code-model training. Original data extraction and visualization supported a hybrid decision framework, demonstrating that optimal training strategies depend jointly on model scale, task abstraction, and deployment context. The review was formatted as an IEEE-style manuscript with the author serving as lead contributor. Finally, these insights informed the design of a deterministic, stateless Adaptive Question Sequencing Algorithm for the YAKSH platform. The engine enforced strict mastery gates (8 unique attempts + 4-correct streak), routed learners through sequential or random practice modes, and exposed a lightweight FastAPI backend with JSON state contracts. A standalone Vite/React demonstration prototype was deployed via Google Cloud Run, while the intended production integration path targeted PostgreSQL under identical API contracts.

The internship produced four principal artifacts: a validated novice error taxonomy, a reproducible open-source LLM evaluation pipeline, an IEEE-formatted literature review on code-LLM training strategy, and a fully documented adaptive sequencing system with frontend demo and backend contracts. Collectively, these contributions advanced explainable, scalable, and pedagogically grounded AI tutoring workflows, providing the YAKSH ecosystem with a deployable architecture for personalized learning progression.

Across all phases, the internship prioritized explainability, reproducibility, and deployment readiness over opaque experimentation. Each deliverable was prepared for practical handoff through documented evaluation scripts, curated datasets, supervisor-facing design briefs, stateless API contracts, and a frontend/backend demonstration workflow that preserved compatibility with the intended PostgreSQL-backed production path. This alignment between empirical evidence, literature synthesis, and engineering execution reduced the gap between research insight and platform integration, positioning the work as both an academic contribution and a concrete implementation strategy for adaptive progression, beginner-safe hinting, and future instructor-facing analytics in open educational technology.

By bridging empirical data analysis, model evaluation, academic synthesis, and full-stack prototyping, this work demonstrated end-to-end technical execution and sustained contribution to open educational technology. Recommended next steps include controlled pilot studies, instructor analytics dashboards, cryptographic state hardening, and hybrid ML-augmented sequencing.

ACKNOWLEDGEMENTS

I thank the FOSSEE team at IIT Bombay for the opportunity to participate in the Semester-Long Internship Programme.

The research and development presented in this report were carried out under the guidance of Prof. Dr. Kushal Shah and Prof. Prabhu Ramachandran. I thank Prof. Dr. Kushal Shah for his supervision and academic guidance throughout the internship.

I thank Ms. Usha Viswanathan and other members of the FOSSEE team for their administrative and project coordination support. I also thank Prof. Vidya Chitre (Head of Department of Information Technology, Vidyalankar Institute of Technology) for permitting my participation in this internship.

I also acknowledge my father, Prof. Dr. Prasad Kulkarni (Fr. C. Rodrigues Institute of Technology, Department of Electronics and Telecommunication) and my mother, Ms. Vidula Kulkarni (Vice Principal, Sanpada College of Commerce and Technology) for their support.

I also thank my family, colleagues, fellow interns, and the readers of this report.

1 INTRODUCTION AND OBJECTIVES

1.1 Background and Context of the Internship

The FOSSEE (Free/Libre and Open Source Software for Education) project at IIT Bombay offered this internship opportunity after a multi-stage screening process involving design, prompt-engineering, and evaluation-oriented assignments. Following selection, I joined the Python Research Team on October 17, 2025, under the guidance of Prof. Dr. Kushal Shah and Prof. Prabhu Ramachandran, with project coordination support from Ms. Usha Viswanathan.

The internship formed part of an effort to improve AI-assisted learning tools for programming education. In particular, the work supported the improvement of *PyPal* by studying student error patterns, evaluating open-source alternatives to proprietary models, and developing adaptive learning workflows. Although the initial internship window was expected to conclude in January 2026, it was extended until April 15, 2026, allowing the work to progress from research and analysis into a software prototype aligned with my Specialization-Based Project requirements.

1.2 Core Objectives and Scope of Work

Over the seven-month internship, the scope of work expanded from exploratory analysis to system design and implementation. The work may be grouped into four major objectives:

- **Objective 1: Student Error Analytics and Behavioral Study**
  Analyze a large confidential dataset from PyPal using `pandas` in order to identify recurring student error patterns, measure error-resolution behaviour across attempts, and convert ambiguous logs into a usable taxonomy of beginner programming mistakes.
- **Objective 2: Open-Source LLM Evaluation and Prompt Design**
  Build an evaluation workflow for open-source code models such as DeepSeek v3.2 Express, Qwen3-Coder, and GPT-OSS, with emphasis on diagnosing the *first* meaningful bug in student code and generating pedagogically safe hints instead of direct solutions.
- **Objective 3: Literature Review and Research Synthesis**
  Conduct a focused literature review on single-language versus multi-language code-model training, synthesize the findings into a structured decision framework, and prepare the study in a formal IEEE-style research format.
- **Objective 4: Adaptive Question Sequencing System Design**
  Design and implement a deterministic backend architecture for the YAKSH coding portal so that question delivery can adapt to student performance through explicit mastery rules, concept-level progression control, and stateless API-driven execution.

1.3 Chronological Narrative and Milestones

- **October 2025: Onboarding, Requirements Gathering, and Data Analytics Initiation**
  Official commencement of the FOSSEE Python (Research in LLM) internship took place on October 17, 2025, followed by orientation and initial scope definition under the guidance of Lead Mentor Prof. Kushal Shah. The primary directive during this period was to explore the confidential PyPal portal dataset and identify pedagogically meaningful patterns in student submissions. A large-scale data-mining workflow was initiated using the Python `pandas` library rather than dashboard-driven tooling, enabling direct inspection of submission histories, student behaviour, and error-resolution patterns. This phase established that students recovered more quickly from simpler syntax errors than from deeper logical or type-related failures.
  During the same period, substantial effort was invested in decoding ambiguous compiler-system

logs and transforming them into an actionable beginner error taxonomy. Through explicit pattern matching and K-means clustering over a numerical representation of the remaining text logs, 1,339 unsuccessful attempts were decoded into 10 distinct error categories. The surviving project documentation did not preserve the exact vectorization routine used to generate those clustering features. A hierarchical sub-categorization of Type Errors further revealed that Function Argument Mismatches — particularly variants such as `main() takes 0 but 1 given` — accounted for 73.9% of all failed coding attempts across the portal. In preparation for the next project phase, error-prone codes were strategically clustered to construct a balanced personal contribution of 50 buggy code snippets for later model evaluation and inclusion in the broader team validation pool.

- **November 2025: AI Model Evaluation, Dataset Lineage, and Multistage Prompting Pivot** In November, the initial benchmarking of the `deepseek/deepseek-v3.2-exp` model was completed on my balanced 50-sample buggy-code contribution, yielding a 100% fix-correctness rate. This evaluation phase also included the administrative closure associated with the API-based experiments. Between November 6 and November 20, major coding work was temporarily paused due to End Semester Examinations. After returning from this academic break, the multistage prompting task was resumed and substantially redefined.
A key architectural insight emerged during this period: because Python execution halts at the first raised exception, a strategy aimed at detecting multiple simultaneous errors was pedagogically and technically unsound. The prompting workflow was therefore pivoted toward identifying and explaining only the *first* occurring error in a stepwise and fully reliable manner. Alongside this prompt redesign, the `dataset_analysis_report.ipynb` workflow was executed to validate dataset lineage mathematically. A data-cleaning engine was implemented to remove hidden spreadsheet artifacts such as `_x000D_`, and a composite key based on Student ID, Question, and cleaned code was generated to verify dataset integrity. My 50-code contribution was then merged into a collaborative team pool of 150 candidate questions, which was programmatically deduplicated on November 4, 2025 to yield the final validation set of 55 unique buggy-code cases; this computational deduplication step was executed by Harshit as part of the team workflow. On November 25, production-ready scripts were executed to create AI-enriched datasets from this 55-case validation set while appending predictive fields such as `AI_Predicted_Error_Type`. Reproducibility was preserved through versioned output batches, `METADATA.json`, and execution tracking in `RUN_LOG.txt`.
- **December 2025: Literature Review Synthesis, Figure Redrawing, and LaTeX Engineering** In December, the internship emphasis shifted more strongly toward comparative literature synthesis on code-LLM training. Extensive research findings were compiled across themes such as negative interference, catastrophic forgetting, and pruning for specialization, establishing a clear depth-versus-breadth trade-off in which single-language training or pruning frequently outperformed multilingual models for narrowly defined tasks. As part of this work, raw numerical data were manually extracted from source papers, including studies such as Athiwaratkun et al. (2023) and Baltaji et al. (2025), and all comparative figures were independently redrawn to avoid plagiarism and improve analytical clarity.
Separate graphs for single-language and multi-language parameter trends were merged into unified comparative charts for clearer interpretation. To support drafting at scale, the LaTeX project was modularized into smaller files and folders so that Overleaf compilation delays could be reduced and collaborative editing could proceed more reliably. This structure was maintained despite formatting disagreements within the team. By the end of the month, the synthesis draft had been submitted with BibTEX support, and a coordination meeting was conducted to delegate pending tasks and circulate detailed Minutes of the Meeting for

alignment.

- **January 2026: Academic Extension, 30B–70B Analysis, and Sequencing Assignment** In January, the final "Data, Case Reports, and Post-Case Analysis on Code LLM Training (30B–70B)" was completed and submitted, satisfying the mid-scale model analysis objective of the research phase. During the same period, a formal request was sent by email to Project Manager Usha Viswanathan seeking an internship extension so that the literature-review work could transition into an implementation-oriented project aligned with the mandatory 2-credit Specialization-Based Project requirement of the college curriculum. Approval was subsequently received confirming that the internship would continue through March 31, 2026.
  This extension enabled a shift from pure research synthesis toward structured implementation planning. A dedicated project coordination channel was established to align academic teammates with internship deliverables. Discussions during this period included screening expectations, UI/UX improvement possibilities, and debugging-prompt generation tasks. On January 27, Prof. Kushal Shah formally assigned high-level project roles to the new group. Exclusive responsibility was assigned for architecting the "Sequencing of Questions for a Beginner," marking the transition from theoretical research contribution into backend software engineering for the YAKSH platform.
- **February 2026: Pivot to Deterministic Question Sequencing and Academic Formatting** In February, project focus shifted toward the adaptive sequencing logic required by the YAKSH portal. On February 11, the backend architecture was formally redirected away from Machine Learning (ML) and Reinforcement Learning (RL)-driven sequencing approaches and replaced with a deterministic, rule-based error categorization engine prioritizing explainability, reproducibility, and low-latency execution. During the same phase, the mathematical rules governing progression were iteratively refined, including the no-soft-unlock mechanism and the final mastery gate requiring an 8-attempt volume floor plus an uninterrupted 4-correct streak before advancement to subsequent programming concepts.
  Ownership was also taken over the supporting curriculum data required by the engine. CSV question pools were created for Concepts 1, 3, and 5, and the base problem sets were deliberately expanded to include five distinct questions for each logical error type so that category-jumping could be supported without immediate repetition under the final 4-streak mastery logic. In parallel, the literature review draft was comprehensively reformatted into a strict two-column IEEE research-paper layout with 10-point font, ensuring compliance with the academic requirements of the internship and associated project submissions.
- **March 2026: Mastery Logic Finalization, API Design, and Project Review Presentation** In March, the core next-question logic engine was formally mapped and implemented into the backend architecture, with a major implementation milestone reached on March 9. Comprehensive flowchart-based documentation was authored in `QUESTION_SEQUENCING _ALGORITHM_SUPERVISOR_BRIEFING.md`, describing the deterministic state-machine transitions between Sequential Mode and Random Mode. The sequencing workflow was simultaneously translated into a stateless API architecture capable of serving adaptive questions without dependence on a heavy local client database.
  The state-tracking design was engineered using structured JSON contracts. Key artifacts included `input.json`, which carried real-time student state such as streaks and attempt counts, and `question.json`, which supported curriculum retrieval and question delivery. On March 13, the "Project Review 1" presentation was delivered to project supervisors, formally introducing the "Novice Progression Wall" and presenting the deterministic adaptive sequencing algorithm as the proposed architectural solution.
- **April 2026: Database Contracts, Full-Stack Prototype Deployment, and Final**

Handoff During the final month of the internship, the backend scope was consolidated through the design of the PostgreSQL schema, migration handling, preparation of `schema_init.py`, and definition of the database contracts needed for longer-term production integration into YAKSH. The stateless API schemas were debugged and stabilized, and the backend codebase was pushed directly to the `master` branch in order to unblock parallel frontend integration work.

To satisfy the final integration requirements of the college project, a fully standalone full-stack prototype of the sequencing system was also constructed. This demo application used Vite with React on the frontend and Firestore for persistence, enabling end-to-end demonstration of progression logic, question delivery, and state synchronization. In contrast, the production YAKSH integration path was designed around PostgreSQL while preserving the same stateless API contracts. The application was subsequently deployed to Google Cloud Run. The extended internship concluded with the formal handoff of the deployed application link and the final packaged repository (`AlgoSequence`) to the project supervisors.

2 PHASE 1: DATA ANALYTICS AND ERROR CATEGORIZATION (OCTOBER 2025 – NOVEMBER 2025)

2.1 Core Objective and Tool Selection

The primary objective of the Data Analytics and Error Categorization project was to analyze the PyPal portal dataset in order to identify student learning patterns, extract pedagogically relevant findings, and systematically decode ambiguous compiler error logs.

At the project's onset, visualization tools such as Tableau or Orange Data Mining were considered; however, Lead Mentor Prof. Kushal Shah advised that Python's `pandas` library was sufficient and emphasized the importance of deeply analyzing the raw data rather than relying on dashboard tools. Consequently, the analytical pipeline was architected in Google Colab using `pandas`, `numpy`, `matplotlib`, and `seaborn`.

2.2 Initial Behavioral Analysis and Error Resolution Rates

The first major deliverable was the notebook `python_pal_analysis_student_performance_SARTHAKKULKARNI.ipynb`, which analyzed success rates, common errors, and student persistence.

- **Learning Persistence:** The analysis tracked how many times students re-attempted a problem after failing, charting attempts from the 1st up to the 10th iteration to build a persistence model.
- **Error Resolution Rate:** A metric was developed to track which error types students were able to fix most successfully in subsequent attempts. The findings revealed that students recovered from simpler errors (such as `NameError` at 45.12% and `ValueError` at 81.16%) much faster than complex `TypeErrors` (36.80%). Following these findings, the analysis was expanded to categorize the thousands of ambiguous "other error type" cases.

2.3 Refined Error Categorization via K-Means Clustering

To unpack ambiguous system logs (such as "RuntimeError", unparsed JSON, or "Failed to Parse Feedback"), a refined categorization engine was implemented in the notebook `ErrorCategorizationAnalysis.ipynb`.

- Explicit string pattern matching and JSON parsing were utilized to extract clearly defined errors from the `exec_feedback` column.
- For the remaining ambiguous feedback, the logs were first converted into a numerical feature representation suitable for **K-means clustering**, after which they were grouped based on text-pattern and structural similarities. The exact vectorization method was not preserved in the surviving project notes.
- This approach decoded the 1,339 unsuccessful student attempts (out of 14,408 total interactions in the dataset) into 10 distinct error categories.

The final distribution of the 1,339 unsuccessful attempts was as follows:

1. **Type Error:** 1,068 occurrences (79.8%)
2. **Index Error:** 70 occurrences (5.2%)
3. **Logic/Output Mismatch:** 69 occurrences (5.2%)
4. **Test Case Failure:** 56 occurrences (4.2%)
5. **Division by Zero:** 31 occurrences (2.3%)

6. **Value Error:** 28 occurrences (2.1%)
7. **Unparsed Feedback:** 7 occurrences (0.5%)
8. **Name Error:** 4 occurrences (0.3%)
9. **Function Call Error:** 3 occurrences (0.2%)
10. **Key Error:** 3 occurrences (0.2%)

2.4 Hierarchical Sub-Categorization of Type Errors

Given that Type Errors accounted for nearly 80% of all failures, a hierarchical sub-categorization was conducted for this group. A flowchart visualization was generated to demonstrate the exact root causes.

- **Function Argument Mismatches** accounted for 92.86% of all Type Errors (989 occurrences).
- 100% of these mismatches manifested as the error message `main() takes 0 but 1 given`.
- This indicated that a single, specific issue—students incorrectly defining their function parameters compared to the platform's expected grading inputs—was responsible for 73.9% of all failed attempts across the entire PyPal platform.

2.5 Dataset Lineage, Data Cleaning, and Deduplication

Prior to using this dataset for AI model evaluation, a validation pipeline was developed and documented in the `dataset_analysis_report.ipynb` notebook.

- **Data Cleaning Engine:** A cleaning function was built to resolve severe "Hidden Character" artifacts, specifically decoding hidden Excel string artifacts (such as `_x000D_`) that were causing strict-matching failures in legacy evaluation systems.
- **Composite Key Generation:** To mathematically prove the integrity and lineage of the filtered data, a Composite Key was generated by concatenating the Student ID, Question ID, and Cleaned Code, demonstrating that the subset of filtered codes was directly derived from the original raw `pypal_data.xlsx` file without corruption.
- **Duplicate Analysis:** A deep audit revealed 1,936 duplicate submissions, which were mapped to distinguish between "System Duplicates" (platform logging errors) and "Submission Duplicates" (where students exhibited "spamming" behavior, submitting the exact same code an average of 4.92 times per question).
- **Generative Proofing:** The generative AI's consistency was evaluated by testing for non-determinism, feeding identical buggy codes through the system multiple times to observe variations in the AI's output.

3 PHASE 2: OPEN-SOURCE LLM EVALUATION AND PROMPTING (OCTOBER 2025 – NOVEMBER 2025)

3.1 Comprehensive Project Execution and Evaluation

3.1.1 Core Objective and Model Selection

The primary objective of the Open-Source LLM Evaluation and Prompting project (Phase 2) was to transition the PyPal platform away from proprietary models (such as GPT-4) toward open-weight systems with a practical mid-scale *active computational footprint* at inference time. The team focused on evaluating **DeepSeek v3.2 Express** (`deepseek/deepseek-v3.2-exp`), alongside other open-weight models such as **GPT-OSS (120B/20B)** and **Qwen3-Coder**. Although DeepSeek v3.2 Express was evaluated as a deployment candidate because it activated roughly 37B parameters per token, it was technically a much larger Mixture of Experts (MoE) architecture with 671B total parameters rather than a dense 30B–70B model. The goal was to build an AI tutor capable of diagnosing errors, generating internal reference solutions, and providing Socratic guidance without revealing the actual code.

3.1.2 Dataset Shortlisting and Evaluation Infrastructure

To create a mathematically sound baseline for evaluating these models, a targeted subset of student data was required.

- **Sample Filtering:** Out of 14,408 total interactions in the PyPal dataset, only 1,339 were identified as unsuccessful attempts (containing errors). From these, I shortlisted a balanced, representative set of **50 distinct buggy code snippets** (stored as `sample_50.csv` or `top_50_balanced_llm_test_samples.xlsx`) as my contribution to the shared evaluation pool. Across the team, 150 candidate questions were then combined and programmatically deduplicated on November 4, 2025, yielding the final set of 55 unique validation cases used for the collaborative pipeline.
- **API and Infrastructure Setup:** To test the models programmatically, OpenRouter API access was provisioned and the required administrative setup was completed through the FOSSEE office.

3.1.3 AI-Enriched Error Dataset Generation Pipeline

A Python pipeline was developed in Google Colab (`Analysis.ipynb`) to automatically generate AI-enriched datasets.

- **Pipeline Execution:** The script processed the collaboratively deduplicated validation set of 55 unique buggy-code rows using the DeepSeek v3.2 Express model, taking approximately 10.79 minutes to complete the run.
- **Temperature Scaling for Specific Tasks:** Temperature controls were used to separate task behaviour: a temperature of `0.0` was used for deterministic error classification, `0.2` for generating functional Python code solutions, and `0.75` for generating an empathetic Socratic tutor response.
- **Data Lineage and Tracking:** The script appended three new columns to the dataset: `AI_Predicted_Error_Type`, `AI_Generated_Solution_Code`, and `AI_Detailed_Response`. To guarantee enterprise-grade reproducibility, the outputs were saved with versioned timestamps to Google Drive, accompanied by a `RUN_LOG.txt` file and an `INPUT_METADATA.json` file incorporating SHA-256 checksums to mathematically prove dataset lineage.

3.1.4 Prompt Engineering and the "Thin Line" Constraint

The prompt engineering phase used pedagogical constraints to ensure the AI acted as a guide rather than a solution provider.

- **Input Separation:** Following supervisor directives, the AI was fed *only* the question and the buggy code for error type prediction (to test its raw diagnostic ability). However, the execution feedback (`exec_output`) was included when generating the detailed Socratic response.
- **The "Thin Line" Pedagogy:** The final `TUTOR_PROMPT_TEMPLATE` established a constraint to provide conceptual feedback without giving away the code solution. The prompt enforced formatting rules: **"No code blocks," "No code suggestions," and a "Maximum 2–3 sentences"**.

3.1.5 Independent Evaluation of DeepSeek v3.2 Express

A dedicated evaluation slide deck titled "Evaluation of DeepSeek v3.2 Express for PyPal Integration" was prepared to document the model's performance. The structured pedagogical evaluation rubric assessed fix correctness, format adherence, bug-identification quality, helpfulness, and tone. Fix correctness measured the model's ability to generate working code, while the debugging quality score evaluated diagnostic clarity, root-cause explanation, and pedagogical safety.

- **Fix Correctness:** The model demonstrated an exceptional **100% fix correctness rate** on complex logical and quadratic problems. The file `debugging_quality_analysis.csv` confirmed a score of 1.0 for `fix_correctness` and `format_adherence` across the test samples.
- **LLM-as-a-Judge Evaluation:** Claude 3.5 Sonnet was utilized as an impartial automated judge to grade the pedagogical safety of the responses. For reproducibility, the judge prompt explicitly instructed: "Evaluate the AI Tutor response for accuracy and beginner-friendliness. Return a JSON object with keys: `Accuracy` (1–5), `Beginner_Friendliness` (1–5), `Acceptability` (true/false), and `Justification` (string)." This ensured that ratings were captured in a structured and auditable format. DeepSeek achieved a **perfect 5/5 rating for 'Helpfulness' and 'Tone'**. NLP analysis revealed the average explanation length was 212 words, utilizing a high count of action verbs and nouns rather than vague fluff.
- **Pedagogical Weaknesses:** Despite perfect code generation, the model's overall debugging quality score was recorded at **0.686**, largely due to a lower bug identification score (0.517) and weaknesses in explicitly defining the root cause in certain edge cases.
- **Final Verdict:** Based on these metrics, I issued a formal "Status: GO" recommendation to project leadership, advising that the DeepSeek v3.2 Express model was highly viable and should proceed to Phase 2 Scale Testing and Pilot User Studies. This executive recommendation reflected both the strong evaluation outcomes and the fact that its Mixture of Experts (MoE) architecture aligned with the project's deployment constraints.

3.1.6 Multistage Prompting and Architectural Pivot

During the evaluation, the team recognized a critical architectural constraint regarding multistage prompting: **Python inherently halts execution at the first exception it encounters**. Consequently, forcing an AI to diagnose "multiple" errors from a single crash log was fundamentally flawed.

- **Strategy Pivot:** Following a consultation with the lead mentor, the multistage prompting strategy was entirely pivoted. Instead of forcing the AI to hallucinate secondary errors, the

multistage pipeline was redirected to **refine the explanation step-by-step and ensure the model correctly identified the *first* error with zero mistakes**. This targeted the specific subset of cases (7 out of 54) where the models initially misdiagnosed the primary failure.

3.1.7 Automated NLP and Linguistic Scoring

To reduce dependence on manual judgment, a set of automated NLP metrics was integrated into the evaluation pipeline using `spaCy` and `textstat`.

- **Readability and Style:** Metrics such as `flesch_reading_ease` (targeting a high score to ensure 8th–9th grade readability for beginners) and `avg_syllables_per_word` were logged.
- **Safety Enforcement:** A critical automated check, `has_code`, was utilized as a boolean red-flag to detect if the LLM violated the core rule by leaking direct code solutions into its textual hint.
- **Socratic Engagement:** The `question_ratio` and `question_count` were tracked to ensure the model was actively asking the student guided questions rather than passively lecturing.

4 PHASE 3: LITERATURE REVIEW ON CODE LLM TRAINING (DECEMBER 2025 – JANUARY 2026)

4.1 Scope and Core Research Objective

The Literature Review on Code LLM Training was conducted as a systematic academic synthesis to evaluate the architectural and performance trade-offs of code-generating Large Language Models. The central inquiry addressed the question: *Does multi-language training weaken single-language specialization?* The review compared single-language (e.g., Python-only) and multi-language training strategies across dense mid-scale models (roughly 30B–70B parameters) and larger architectures, including AlphaCode, CodeLlama, DeepSeek-Coder, StarCoder, and CodeGen. Where relevant, the review also distinguished between dense parameter count and active inference footprint in modern sparse MoE systems so that models such as DeepSeek v3.2 Express were not conflated with dense 30B–70B designs.

4.2 Research Methodology and Comparative Analysis

Transitioning from empirical evaluation to academic synthesis, the third major objective was to conduct a systematic literature review. The review established that while training on multiple programming languages grants broad capabilities, it also introduces risks such as "negative interference" and "catastrophic forgetting," which can weaken a model's specialized depth in a single language.

To systematically analyze how modern models handle this trade-off, a comparative structure was established. I completed the Phase 1 "Literature Mapping" and Phase 2 "Analysis of Training on Multiple Programming Languages" specifically for the DeepSeek V3.2 Express model, and provided this structured directory as a baseline template for the rest of the team to replicate for Qwen3-Coder and GPT-OSS. A shared Google Spreadsheet was used to compile literature review sources consistently. On December 8, the initial literature survey was submitted, broken down into three documents focusing on negative interference, pruning for single-language performance, and multilingual training methodologies.

As the review matured, the analysis was organized around a more rigorous narrative arc: controlled empirical evidence for single-language and multi-language training, scaling-law interpretations, and industrial case studies spanning systems such as AlphaCode, CodeLlama, DeepSeek-Coder, StarCoder, and CodeGen. This structure made it possible to compare not just benchmark outcomes, but also the practical design logic used in production systems, where multilingual pretraining repeatedly appeared as the foundation and monolingual specialization appeared later as a targeted optimization.

4.3 Original Data Extraction and Visualization

A project requirement was that no figures from external research papers could be directly copied into the review. To address this, a manual data extraction pipeline was implemented.

- **Raw Data Extraction:** Instead of using computational compression or algorithms, raw numerical values were manually extracted directly from the tables of leading research papers, specifically *Athiwaratkun et al. (2023)* and *Baltaji et al. (2025).*
- **Original Visualizations:** Utilizing this raw data, completely original preliminary figures were generated to compare monolingual and multilingual training performance.
- **Graph Merging:** Following supervisor feedback, these visualizations were refined by merging the separate graphs for single-language and multi-language training parameters into unified

comparative charts, allowing for clear, side-by-side analysis on the same benchmarks. The updated literature review draft with these merged graphs was submitted on December 24.

The figure work later expanded beyond simple mono-vs-multi comparisons into scaling-law and data-composition visual analysis. This included reconstructing plots showing multilingual token-allocation effects, cross-language transfer behavior, and the trade-off between pure programming-language corpora and mixed natural-language plus code corpora. These additions strengthened the review by connecting literature claims to interpretable visual evidence rather than leaving them as isolated textual summaries.

4.4 Document Expansion and LaTeX Modularization

As the research expanded from analyzing 2 papers to 20 papers, the structure of the document required significant refactoring.

- **Expanding Section 3:** Section 3 ("Evidence from the Literature") was rewritten and expanded. Stronger comparative studies were integrated and the discussion was restructured to clearly define the exact conditions under which single-language training is preferable versus multi-language training.
- **LaTeX Project Modularization:** To streamline team collaboration and prevent compilation timeouts on Overleaf, the LaTeX project directory was completely modularized. The monolithic `main.tex` file was broken down, creating separate, organized files and folders for each individual section, figure, and table. When a teammate attempted to merge the structure back into a single file, the modularized structure was retained to ensure final integration and compilation remained stable.

4.5 Mid-Scale Model Analysis (30B–70B Parameters)

To target the specific architectural tier of interest, a focused review of dense mid-scale code language models (approximately 30B–70B parameters) was conducted. On January 12, 2026, a data and post-case analysis report was submitted covering models like CodeLlama-34B/70B, AlphaCode, DeepSeek-Coder-33B, and StarCoder. This report synthesized common training strategies, recurring trade-offs, and the constraints that shaped language selection at this parameter scale. DeepSeek v3.2 Express was treated separately in the broader discussion as a sparse MoE comparator: despite an inference footprint of about 37B active parameters, its full architecture is substantially larger at 671B total parameters.

4.6 Final Formatting and Academic Integration

Recognizing the depth of the literature review, formal permission was requested from the supervisor and administration to transition this research into a structured implementation project to satisfy a mandatory 2-credit "Specialization-Based Project" requirement for the college curriculum.

Following review, the literature review was reformatted from a standard thesis-style layout into a two-column IEEE research-paper format with 10-point font size. On March 31, 2026, the final formatted version of the literature review paper (`main.pdf`) was submitted alongside the GitHub repository link, concluding Phase 3. Because I led the core drafting, synthesis, figure reconstruction, and final IEEE formatting of this manuscript, this body of work established my role as the first author of the literature review paper.

4.7 Key Theoretical Findings and Decision Framework

The review established that neither strategy is universally superior; the optimal approach is governed by a hybrid paradigm determined by three measurable factors:

- **Model Scale:** A consistent crossover threshold exists between 672 million and 2.7 billion parameters. Below 1B parameters, single-language training is superior due to negative interference and catastrophic forgetting. Above 2B parameters, multi-language training enables positive transfer and data spillover, achieving +1.7 to +2.9 percentage points improvement even on single-language tasks.
- **Task Abstraction:** Syntactic tasks (precise code repair/completion) favor single-language training, showing +16% BLEU-4 improvement. Semantic tasks (translation, search, clone detection) favor multi-language training, with +14–35% improvement in zero-shot cross-lingual scenarios.
- **Deployment Context:** All major industrial systems use a hybrid approach: multilingual pre-training for broad capability, followed by single-language fine-tuning for precision. This approach yields +21 percentage points F1 improvement in industrial code review deployments.

An additional conclusion from the deeper synthesis was that training-data composition matters alongside language count. Evidence from CodeGen-style studies showed that programming-language-only corpora produce the strongest pure code-generation performance, while mixed natural-language plus code corpora sacrifice some benchmark accuracy in exchange for prompt understanding, explanation ability, and conversational utility. This nuance helped refine the final decision framework: multilingualism is only one axis of design, and optimal model behavior also depends on how training capacity is divided between code tokens, natural language, and high-synergy language pairs.

Table 1: Decision framework for code-LLM training strategy synthesized from the Phase 3 literature review

Phase 3 Decision Framework for Code-LLM Training Strategy

Model Scale	Task Type	Recommended Strategy
< 1B parameters	Syntax-heavy repair/completion	Single-language pretraining
672M–2.7B crossover zone	Mixed workloads	Hybrid evaluation required
> 2B parameters	Semantic transfer/search/translation	Multi-language foundation
Any scale	Organization-specific deployment	Monolingual fine-tuning on top

Interpretation: the literature supported a hybrid decision rule rather than a universal winner.

Key Quantitative Findings from Literature

The following tables summarize the main quantitative results that informed the decision framework. Table 2 demonstrates the scale-dependent crossover effect, showing that multi-language training becomes superior at 2.7B parameters and above. Table 3 illustrates the substantial gains in cross-lingual transfer capabilities, particularly for unseen languages. Table 4 reveals that while

single-language training remains optimal for clone detection (0.92 F1), multi-language training achieves 90% of that performance (0.83 F1) while enabling cross-lingual capabilities. Table 5 shows that multi-language models are more robust to surface-level variations, suffering 22% less degradation under semantic-preserving transformations.

Table 2: Pass@1 by model size for single-language and multi-language training

Model size	Single (%)	Multi (%)	Gain (pts)
2.7B	24.1	27.0	2.9
13B	33.6	35.3	1.7

Table 3: Zero-shot translation Pass@1 for single-language and multi-language training

Target	Single (%)	Multi (%)	Gain (%)
Java	36.0	41.0	14.0
JavaScript	33.0	38.0	15.0
PHP	20.0	27.0	35.0
Ruby	15.0	20.0	33.0
Kotlin	14.0	18.0	29.0

Table 4: Clone-detection performance (F1) for single-language and multi-language training

Setting	Single (F1)	Multi (F1)	Zero-shot (F1)
Clone detection	0.92	0.83	0.49

Table 5: Pass@1 drop after semantic-preserving transformations

Metric	Single drop (%)	Multi drop (%)	Resilience gain (%)
Average drop (In-domain)	9.8	7.6	22.0
Average drop (Out-of-domain)	9.0	8.0	11.0

4.8 Document Architecture and Team Coordination

The review expanded from 2 to 20 papers, requiring modularization of the LaTeX project (e.g., `evidence_single.tex`, `evidence_multi.tex`, `alphacode_case_study.tex`). Section 3 was rewritten to clarify conditions for negative interference vs positive transfer. Standardized templates were provided for team-wide consistency.

5 PHASE 4: ADAPTIVE QUESTION SEQUENCING ALGORITHM (FEBRUARY 2026 – APRIL 2026)

5.1 Problem Statement and Architectural Pivot

The YAKSH coding platform previously utilized a static, one-size-fits-all question delivery sequence where students advanced regardless of their conceptual mastery. This approach often led to student frustration, termed the "Novice Progression Wall." In February 2026, I was assigned to develop an adaptive question-sequencing logic to replace this static system.

The sequencing engine was shifted away from Machine Learning (ML) and Reinforcement Learning (RL) approaches toward a deterministic, rule-based error categorization system. This change was driven by the need for explainability, reproducibility, and low-latency deployment in an educational setting.

5.2 Core Algorithmic Design

The Adaptive Question Sequencing Algorithm was engineered as a deterministic, rule-based tutoring engine that dynamically adapted to individual student pacing and enforced conceptual mastery before allowing progression.

5.2.1 Linear Concept Progression

Learning was structured into primary Concepts (e.g., Conditionals → Loops → Dictionaries). Students had to complete these in a strict linear order. No soft unlock was permitted: learners could not access subsequent concepts until both mastery gates were satisfied, preventing premature progression.

5.2.2 Intra-Concept Sequencing and Error Categories

Within each concept, questions were categorized by predefined logical error types (e.g., Syntax, Conceptual, Input, Variable, Loop, etc.). The very first question a student saw in any new concept was always a Syntax question to test foundational structure.

5.2.3 Mastery Gates

To master a concept and unlock the next one, the student had to satisfy two strict conditions simultaneously at the moment of progression:

Gate 1: **The Volume Floor.** Learners had to complete at least 8 unique attempts. Retries or forced re-solves within the same error type did not increase this count; only newly served questions contributed toward the volume floor.

Gate 2: **The Mastery Key.** Learners had to achieve an uninterrupted streak of 4 correct answers in a row.

Advancement was granted only when both conditions were satisfied at the same time. In practice, the learner had to first accumulate at least 8 unique attempts and then hold a 4-answer correct streak. There was no partial promotion for meeting only one gate.

5.2.4 Strict Evaluation and Forced Re-Solves

If a student answered incorrectly or relied on an AI-generated hint, the streak was instantly reset to 0. To prevent students from skipping weak areas, the algorithm then forced a re-solve on a different question from the same error category.

5.2.5 System States: Sequential vs. Random Mode

The algorithm dynamically routed students through different operational states based on their performance. These states were mutually exclusive for any given request, so the learner was always in exactly one mode at a time:

- **Sequential Mode:** The default state upon starting a concept. The engine moved down the prioritized list of error categories step-by-step. If a student answered correctly without hints, they advanced to the next error type, ensuring diverse practice.
- **Random Mode:** This mode was triggered if a student reached the volume floor of 8 attempts but failed to meet the required 4-correct streak (Attempts ≥ 8 AND Streak < 4). In Random Mode, the engine pulled questions randomly from any error type—strictly avoiding the immediate repetition of the previous error type—and continued serving them until the 4-streak mastery was achieved.

5.3 Question Pool Engineering and Curriculum Design

To fuel the sequencing algorithm, targeted CSV question pools were authored and curated for Concepts 1, 3, and 5 (Conditionals, Loops with Lists, and Dictionaries). The base problem set was expanded by strategically adding 30 extra questions (5 distinct questions for each specific logical error type). This distribution was mathematically necessary to ensure sufficient question volume to support category-jumping without immediate repetition under the final 4-streak mastery logic.

A more explicit internal error taxonomy was also implemented in the prototype codebase, with ordered categories such as Syntax, Conceptual, Input, Variable, Array/String, Condition, Loop, Computation, Branching, and Output. This fixed ordering was important because sequential mode depended on a stable error-type index, while random mode needed to exclude the immediately previous error type and still preserve meaningful pedagogical coverage. The question-selection service therefore had to support local pool filtering, unseen-question filtering, recent-question exclusion, and fallback behavior when a specific error-type pool was exhausted.

5.4 Technical Architecture and Implementation

The primary technical responsibility was architecting the backend infrastructure to serve the adaptive questions seamlessly.

5.4.1 Stateless API Design

The backend engine was engineered to be exceptionally lightweight and stateless, eliminating the need for a heavy local database on the YAKSH client. All tracking state information (streaks, attempts, current error type) was maintained in structured JSON and transmitted through the API either in the request body or through the `X-Student-State` header as base64-encoded state snapshots. In the documented prototype, these payloads were encoded for transport but were not described as cryptographically signed; accordingly, protection against client-side tampering was best treated as a future hardening step through signed or otherwise cryptographically validated state payloads. This interface design allowed the same sequencing logic to support both internal evaluation scripts and external frontend clients while keeping the engine deterministic and portable.

The underlying state schema tracked not only the current concept and streak, but also the current error-type index, forced-resolve status, random-mode status, last-question metadata, recently presented question identifiers, and a state-version field for compatibility. In practice, this made the system behave like a deterministic state machine: every request encoded the learner's

full progression context, and every response returned an updated state that the client could immediately reuse.

This architecture was also intentionally designed to interoperate with the broader YAKSH AI-assistance stack presented during integration reviews. In the combined workflow, the sequencing backend could supply the selected buggy code context, concept metadata, and exact error-type labels to the separate AI-driven Persona Feedback wrapper (including the HC Verma persona module developed by teammates Rudrapratap and Manini), allowing the LLM layer to generate pedagogically styled feedback on top of a deterministic question-selection pipeline.

5.4.2 PostgreSQL and Schema Management

A persistent relational design was also prepared for integration into the larger YAKSH platform. The proposed PostgreSQL schema separated `StudentState`, `StudentProgress`, `Question`, and `Submission` responsibilities so that progression logic, mastery status, question metadata, and audit-trail submissions could be maintained cleanly. This database design was intended to preserve analytics and integrity even though the sequencing logic itself remained stateless at the API layer.

In parallel, a standalone demonstration prototype validated the same ideas using Firestore collections for questions, per-concept user progress, and submission history, with a Vite/React frontend for interactive validation. This alternate implementation was explicitly a demo environment rather than the intended production storage layer. The production YAKSH integration targeted PostgreSQL, but preserved the same stateless API contracts and progression fields. This showed that the sequencing model was portable across storage layers: whether backed by PostgreSQL in the production path or Firestore in the demo application, the same core fields—attempt count, streak, current mode, error-type index, recently seen questions, and submission history—were sufficient to drive adaptive progression and frontend synchronization.

5.4.3 Deployment Efficiency and Scalability

Because the system utilized deterministic numerical thresholds rather than ML inference, compute overhead was minimal. This design kept the platform suitable for low-cost and scalable institutional deployment. Additional middleware was also designed for request/response logging, stateless endpoint monitoring, and future metrics collection, allowing the backend to remain operationally observable without changing the sequencing rules themselves.

5.4.4 Backend Integration and Debugging

Backend debugging was handled, the GET and POST API schemas were updated to remove deprecated keys, and the stabilized backend codebase was pushed directly to the `master` branch to unblock the frontend integration process. Endpoint support was extended beyond simple next-question and submit-answer flows to include reset and recovery operations, explicit mastery signalling in submission responses, and graceful handling of exhausted question banks. These changes were important during integration because the frontend had to distinguish between ordinary progression, concept mastery, invalid submissions, and no-content states without relying on hidden server-side assumptions.

5.5 Development Milestones and Documentation

The engineering of this system progressed through structured sprints:

- **February 11, 2026:** Initial pivot from ML frameworks to a rule-based error categorization architecture.
- **February 23, 2026:** Finalization of the mathematical thresholds, the 'no soft unlock' rule, and the concept-mastery gate.
- **March 9, 2026:** Formal implementation of the core next-question logic engine and state-routing.
- **March 16, 2026:** Finalization of the stateless API header structure and backend testing.
- **Late March–April 15, 2026:** Frontend integration, standalone prototype completion, deployment, and final project handoff.

The accompanying project documentation evolved beyond a high-level proposal into an implementation-oriented specification. It formalized state fields, transition rules, mode switching, database entities, and API contracts, including the exact condition for entering Random Mode (Attempts ≥ 8 and Streak < 4), the post-submission mastery condition (Attempts ≥ 8 and Streak ≥ 4), and the rule that forced-resolve mode was activated on incorrect submissions or hint usage. Edge cases such as inconsistent state, repeated question handling, and question-bank exhaustion were explicitly documented so the sequencing engine could be debugged and handed off with predictable behavior.

Flowchart-based documentation was authored to describe the algorithm's lifecycle, including `QUESTION_SEQUENCING_ALGORITHM_SUPERVISOR_BRIEFING.md`. The logic and integration rules were also documented in files such as `ADAPTIVE_SEQUENCING_PROPOSAL.md` and `TECHNICAL_IMPLEMENTATION_BRIEF.md`, which served both as supervisor-facing design notes and as implementation references for backend/frontend integration.

5.6 Project Review and Final Deployment

On March 13, 2026, I represented the team and delivered the "Project Review 1" presentation to the project supervisors. The solution to the "Novice Progression Wall" was formally presented, detailing the deterministic question sequencing algorithm, the mastery gate logic, and the stateless API integration. During this review, the sequencing backend was positioned as the control layer that would feed curated buggy-code context and precise error-type information into the parallel "Innovation 2: AI-Driven Persona Feedback" component, so that persona-based tutoring responses could remain grounded in the learner state and remediation target chosen by the sequencing engine.

For the final college presentation and project handoff, a standalone, full-stack demonstration prototype was constructed utilizing Vite with React for the frontend interface and connected to a Firestore backend database. The prototype included concept start/reset flows, progress retrieval, next-question fetching, submission logging, mastery celebration behavior in the user interface, and persistence of submission histories for later review. This Firestore + React stack was used specifically for demonstration and validation; the production YAKSH integration path instead targeted PostgreSQL with the same stateless API contracts described earlier. The platform was containerized and deployed to Google Cloud Run, demonstrating that the sequencing workflow was not merely theoretical backend logic but an end-to-end deployable learning application. A recorded project demonstration for this Phase 4 system was also made available via `YouTube Live demo`.

In April 2026, the deployed application link and the completed repository (`AlgoSequence`) were officially handed off to the supervisor, concluding the backend architecture contributions for the internship.

6 TECHNICAL CONTRIBUTIONS

6.1 Backend: Sequencing Engine and Stateless API

- Developed a deterministic sequencing engine prioritizing error targeting, mastery gating, and reproducibility.
- Implemented stateless API endpoints (`/next-question/stateless`, `/submit-answer/stateless`) supporting both body and header-based state passing.
- Designed request/response schemas and error handling for integration with frontend and evaluation drivers.
- Structured the backend interfaces so that selected buggy-code context, learner state, and exact error-type labels could be passed onward to the AI Persona wrapper, enabling deterministic sequencing to work in tandem with persona-based LLM feedback during end-to-end YAKSH integration.
- See: `api_architecture/engine/sequencing_engine.py` and `api_architecture/api/routes.py`

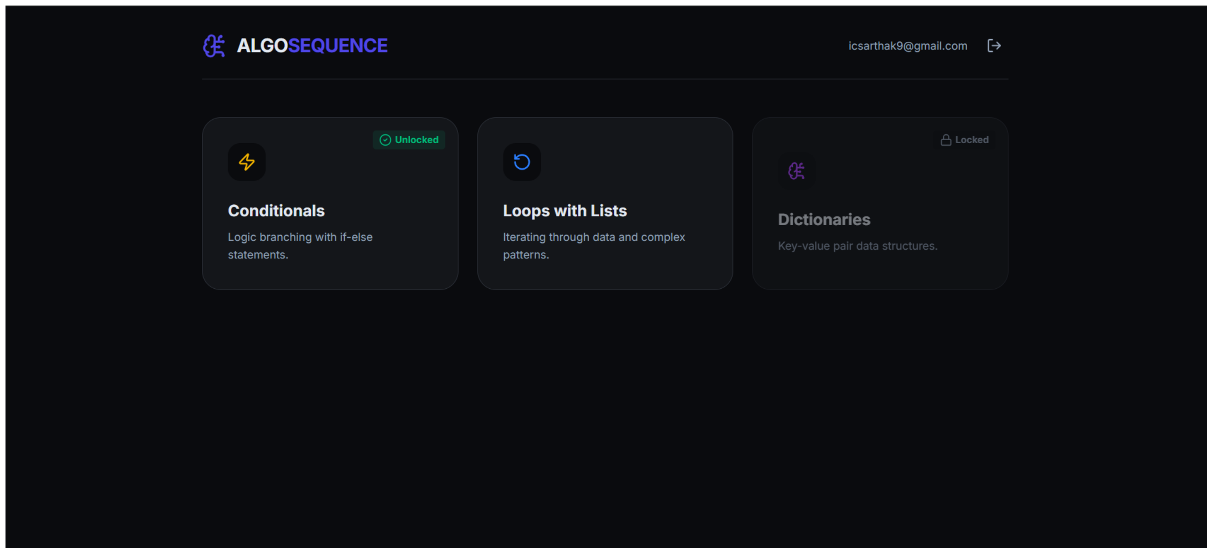
6.2 Error Categorization and Tutor Prompting

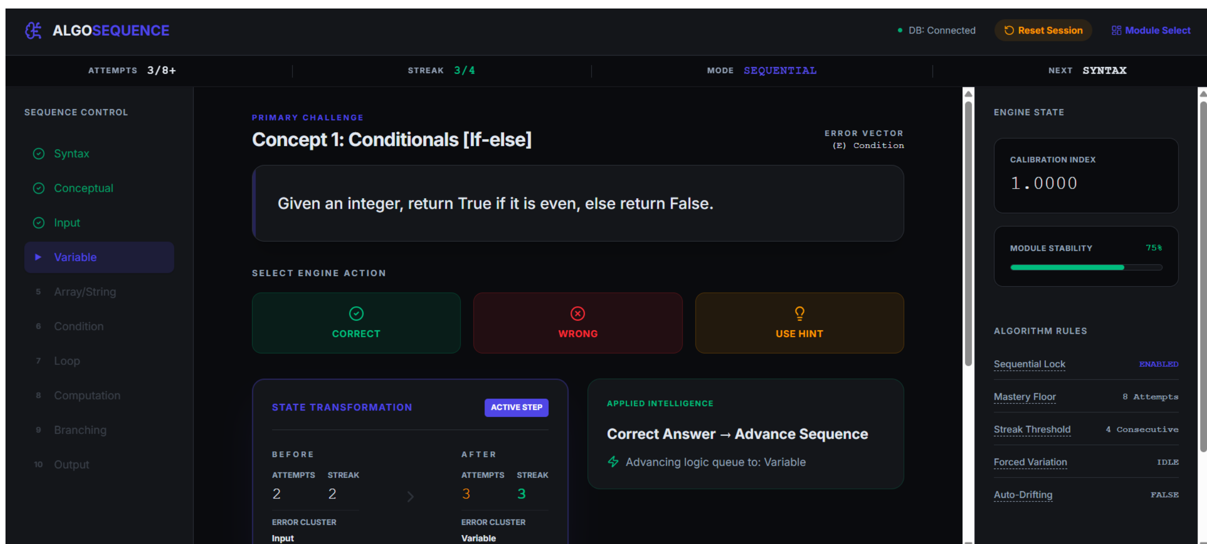
- Built an operational error taxonomy for Python programming mistakes.
- Implemented a graduated hinting pipeline using LLMs, with prompt templates and hint-level policies.
- Integrated hinting with the sequencing engine and simulated hint efficacy.
- See: `prompt_templates/`, `error_benchmark.pdf`

6.3 Frontend Demo and Integration

- Developed a standalone Vite/React frontend demo visualizing student progression, state, and API integration.
- Supported Google authentication and optional Firestore persistence for demo use, while keeping the same stateless API contracts intended for PostgreSQL-backed YAKSH deployment.
- Enabled rapid prototyping and reproducibility for demo sessions.
- See: `src/App.tsx` and `src/firebase.ts`

Figure 1 shows representative frontend views used during integration and demo validation. These screens highlight the student-facing workflow, progression logic, and end-to-end connectivity between the standalone React demonstration interface and the backend sequencing API that is intended to remain unchanged for PostgreSQL-backed YAKSH integration.





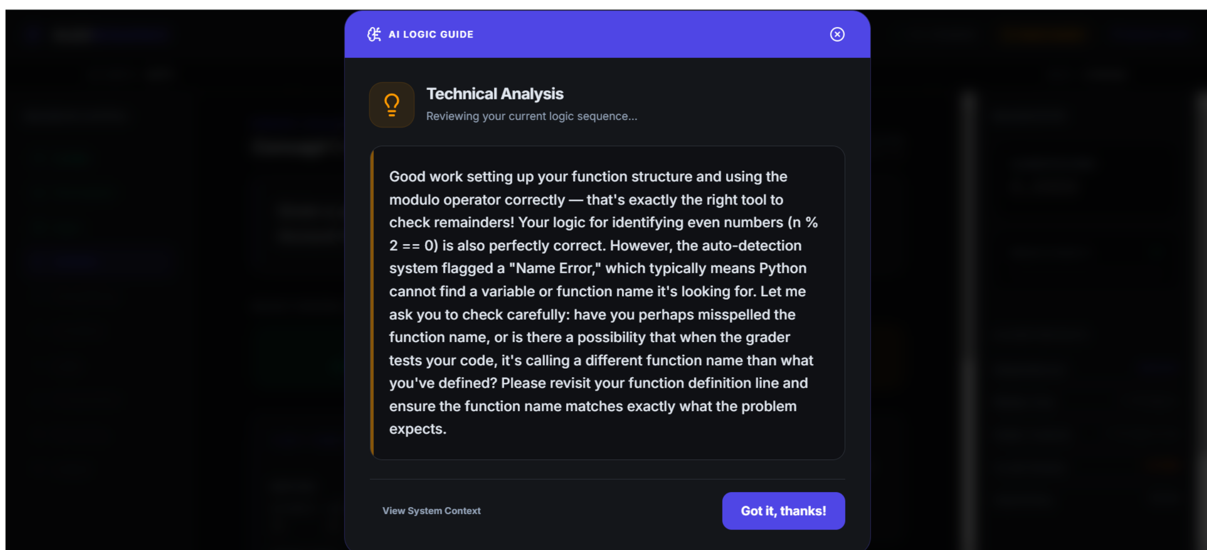


Figure 1: Frontend demo screens showing the standalone Firestore + React demonstration workflow, question delivery, and state-aware progression during Phase 4 development.

6.4 Data, Evaluation, and Reproducibility

- Curated and converted question banks (CSV → JSON) for multiple concepts and error types.
- Built a stateless driver to simulate sessions and generate analysis-ready logs (`output.json`).
- Provided scripts and instructions for reproducing evaluation metrics (questions-to-mastery, mastery rate, error coverage).
- See: `api_architecture/data/questions.json` and `api_architecture/demo_generate_output.py`

7 CROSS-PHASE INSIGHTS AND DESIGN LESSONS

The later stages of the internship were not isolated deliverables; rather, the literature review, LLM evaluation, and sequencing-system implementation progressively informed one another. The following synthesis captures the main cross-phase design insights and practical lessons that shaped the final system.

7.1 Research-Grounded Design Rationale

The literature review and model-evaluation phases helped clarify which architectural principles were most appropriate for an educational coding-support system. The most important takeaways were:

- **Explainability over opacity:** The literature and internal evaluation both supported the use of transparent, auditable system behavior in educational settings, which reinforced the eventual shift toward deterministic sequencing rather than opaque ML-driven progression logic.
- **Hybrid model strategy:** The research synthesis showed that multilingual foundations with targeted specialization remain the dominant industrial pattern, while the model-evaluation phase showed that practical deployment decisions must also account for active inference footprint, pedagogical safety, and controllability.
- **Metric-driven engineering:** The evaluation work demonstrated the importance of explicit metrics such as fix correctness, format adherence, bug-identification quality, and pedagogical tone. These findings directly influenced the decision to keep the Phase 4 sequencing engine measurable, reproducible, and easy to validate.
- **System interoperability:** Both the research and implementation phases highlighted that lightweight, well-defined interfaces are essential when multiple project components must work together, including question sequencing, frontend delivery, and persona-based AI feedback.

7.2 Implementation Challenges and Lessons Learned

The implementation and handoff process revealed several practical lessons that cut across the internship as a whole:

- **Dataset lineage matters:** Evaluation claims became credible only when sampling, collaborative deduplication, metadata tracking, and versioned outputs were handled carefully and reproducibly.
- **Pedagogical safety requires restraint:** Making an AI tutor genuinely helpful without letting it leak full answers required repeated prompt refinement, controlled response formatting, and deliberate separation between diagnosis and explanation tasks.
- **Stateless design simplifies integration but shifts responsibility:** The stateless API made the backend easier to integrate with external clients and wrapper modules, but it required more careful state validation, contract design, and recovery-path handling.
- **Mastery thresholds shape learner experience:** Finalizing the 8-attempt floor and 4-correct streak showed that small changes to mastery rules can significantly alter progression behavior, remediation pressure, and question-pool requirements.
- **Research-to-product translation is multi-layered:** Converting analytical findings into a deployable prototype required aligning theory, data curation, backend logic, frontend behavior, documentation, and supervisor-facing communication rather than treating them as separate tracks.

8 CONCLUSION

8.1 Summary of the Internship Journey

The internship progressed through four clearly defined phases, each contributing a distinct layer of research, analysis, and implementation to the broader objective of improving AI-assisted Python learning workflows. In **Phase 1**, large-scale PyPal student-interaction data were analyzed, ambiguous error logs were cleaned, and an interpretable taxonomy of beginner programming mistakes was developed. This phase established the empirical foundation of the internship and demonstrated that Function Argument Mismatches — especially variants such as `main() takes 0 but 1 given` — constituted the dominant error cluster, accounting for 73.9% of the observed failures.

In **Phase 2**, this analytical foundation was extended into open-source LLM evaluation and prompt design. The work focused on identifying the first meaningful bug in student code, preventing overhelpful answer leakage, and generating pedagogically safe hints. This phase produced a structured evaluation workflow, prompt templates, internal scoring criteria, and practical evidence that open models such as DeepSeek v3.2 Express could support tutoring-oriented debugging tasks under controlled prompting constraints.

In **Phase 3**, a focused literature review was conducted on single-language versus multi-language training strategies for code models, and the findings were converted into a formal IEEE-style paper. This phase strengthened the theoretical grounding of the internship by synthesizing research on scaling behavior, cross-lingual transfer, syntax sensitivity, and hybrid training strategies. It also connected the implementation choices of the internship to a broader design rationale centered on explainability, reproducibility, and model-task fit.

In **Phase 4**, the earlier analytical and research findings were translated into software design and deployment through the development of a Stateless Deterministic Question Sequencing Algorithm for the YAKSH platform. This phase included mastery-gated progression rules, error-category-aware question delivery, stateless API design, question-pool engineering, frontend integration, and both (i) a standalone Firestore + React demonstration prototype and (ii) a production integration path targeting PostgreSQL with the same stateless API contracts. Taken together, the four phases demonstrated a progression from data analysis, to model evaluation, to research synthesis, and finally to end-to-end system implementation.

8.2 Impact on the YAKSH Ecosystem

The internship contributed both research insight and deployable infrastructure to the YAKSH ecosystem. The early error-analysis work clarified which student failure modes were most pedagogically significant, while the LLM evaluation phase examined how AI-generated assistance could remain diagnostically useful without becoming solution-revealing. The literature review further supported the selection of transparent and reproducible system design over opaque experimentation in an educational setting.

The most direct platform contribution was the adaptive sequencing system developed in the final phase. This work replaced static question ordering with a rule-based progression framework that responded to student performance through explicit mastery gates, forced re-solves, and state-aware question selection. By combining deterministic logic with stateless API contracts and reproducible question delivery, the project established a practical path toward personalized learning support that remained explainable, testable, and suitable for institutional deployment. The standalone Firestore + React application demonstrated the workflow end to end, while the intended production YAKSH architecture remained PostgreSQL-backed under the same stateless

contract model.

8.3 Academic Fulfillment and Final Handoff

Beyond its value to the FOSSEE project, the internship also fulfilled the requirements of a 2-credit Specialization-Based Project for TE Semester VI. The completed work included data-analysis notebooks, an error taxonomy, evaluation and prompt-design workflows, a literature review paper, backend sequencing logic, a frontend demonstration interface, reproducible evaluation scripts, PostgreSQL-oriented production integration contracts, and implementation-oriented documentation. These artifacts collectively reflect both academic engagement and applied engineering execution across the full internship period.

At the close of the internship, the completed repository (`AlgoSequence`), deployed demonstration prototype, production-integration documentation, and supporting materials were organized for handoff to the supervising team. This final handoff marked the conclusion of a seven-month effort that combined research, software development, experimentation, and technical communication into a coherent body of work.

8.4 Next Steps and Future Work

The internship established a strong base for continued development in both research and implementation. The following next steps are recommended to extend the work:

- **Pilot Studies:** Conduct controlled studies with real students to measure how adaptive sequencing and guided hinting affect learning outcomes, persistence, and time-to-mastery.
- **Persistent Analytics:** Add long-term progress storage, instructor dashboards, and cohort-level analytics so that concept mastery and intervention needs can be tracked over time.
- **Question Pool Expansion:** Expand the curated question banks across all YAKSH concepts and strengthen error-type coverage so the sequencing engine can be deployed at broader curricular scale.
- **State Integrity Hardening:** Add cryptographic signing or equivalent validation for `X-Student-State` and other client-carried state payloads so that streaks, attempts, and progression flags cannot be tampered with client-side.
- **Hybrid Sequencing Research:** Explore future extensions in which the deterministic engine remains the core progression layer while ML-based components assist with difficulty estimation, hint personalization, or learner-risk prediction.

REFERENCES

References

[1] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, *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] R. Baltaji, S. Pujar, L. Mandel, M. Hirzel, L. Buratti, and L. R. Varshney, "Cross-lingual transfer in programming languages: An extensive empirical study," *Transactions on Machine Learning Research*, Jun. 2025.

[4] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training LLMs on programming and natural languages," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[5] A. Kurz *et al.*, "Data curation strategies for code language models," arXiv preprint arXiv:2410.12456, 2024.

[6] M. Puccioni *et al.*, "Resource-efficient training of programming language models," in *Proceedings of the International Conference on Machine Learning (ICML)*, to appear, 2025.

[7] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[8] B. Rozière *et al.*, "Code Llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.

[9] J. Yang, S. Guo, L. Jing, W. Zhang, A. Liu, C. Hao, Z. Li, W. X. Zhao, X. Liu, W. Lv, and B. Dai, "Scaling laws for code: Every programming language matters," arXiv preprint arXiv:2512.13472, 2025.

[10] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, *et al.*, "StarCoder: May the source be with you!," arXiv preprint arXiv:2305.06161, 2023.

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn program synthesis," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[12] J. Schrittwieser *et al.*, "Competitive programming with AlphaCode," presented at MAPS 2022 (The 6th Annual Symposium on Machine Programming), PLDI 2022, 2022.

[13] DeepMind, "AlphaCode 2 technical report," technical report, 2023. Describes evolution from AlphaCode to AlphaCode 2 with Gemini-based improvements.

[14] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, *et al.*, "Llama 2: Open foundation and fine-tuned chat models," arXiv preprint arXiv:2307.09288, 2023.

[15] Z. Chen *et al.*, "AlchemistCoder: Harmonizing and eliciting code capability by hindsight tuning on multi-source data," in *Proceedings of the 38th Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[16] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, *et al.*, "DeepSeek-Coder: When the large language model meets programming—the rise of code

intelligence," arXiv preprint arXiv:2401.14196, 2024.

[17] J. Jiang *et al.*, "A survey on large language models for code generation," arXiv preprint, 2024.

[18] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, *et al.*, "StarCoder2 and The Stack v2: The next generation," arXiv preprint arXiv:2402.19173, 2024.

[19] D. Kocetkov *et al.*, "The Stack: 3TB of permissively licensed source code," in *Proceedings of EMNLP*, 2022.

[20] F. Cassano *et al.*, "A survey on LLM-based code generation for low-resource and domain-specific programming languages," arXiv preprint arXiv:2311.07989, 2023.

CODEBASE LINKS

- **Coding Interface Prototype Development:** `Open GitHub repository`
- **Literature Review (Mono vs Multi-Lingual):** `Open GitHub repository`
- **AlgoSequence (Adaptive Sequencing Engine):** `Open GitHub repository`
- **Phase 4 Project Demo Video:** `Open YouTube demo`