



UNIVERSITY OF
LEICESTER

School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report

An AI Multiple-Choice Programming Exercises Tutor

Sai Raghava Vemuri

Srv10@student.le.ac.uk

249034443

Project Supervisor: Dr. Wentao Li

Principal Marker: Dr. Martina Palusa

Word Count: 9900

05-09-2025

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Sai Raghava Vemuri

Date: 05-09-2025

ABSTRACT

Multiple-choice questions (MCQs) remain a central tool in programming education, yet traditional static item banks fail to adapt to individual learners' progress. This project presents the design and implementation of an **AI Programming MCQ Tutor** that combines a curated dataset, large language models (LLMs), and an adaptive difficulty engine to deliver personalised practice with reliable quality and low latency.

The system integrates three core components. First, a curated dataset of 100 labelled MCQs per language (Python, Java, C) provides both a seed for AI-driven generation and a trusted fallback. Second, an AI pipeline powered by the OpenAI API generates new questions at the same, higher, or lower difficulty levels. Each candidate is checked against a strict schema and validated independently to ensure exactly one correct answer. Any invalid or ambiguous generation triggers an immediate fallback to the curated dataset, guaranteeing continuity. Third, an adaptive engine functions as a transparent state machine: for each example MCQ, it produces two questions (one same-level and one conditional), tracks learner performance and streaks, and proposes difficulty adjustments with explicit user confirmation.

The web application is implemented with **Flask** and server-rendered HTML/CSS, backed by **MySQL on Railway** for persistence. A concurrency design prefetches the next item during feedback review, making quiz progression feel instant while eliminating race conditions. The frontend includes login, signup, password reset, quiz, result, and dashboard views, all styled for readability and accessibility.

Evaluation followed a hybrid strategy suited to the non-determinism of LLM outputs. Across three languages and all difficulty levels, the generator alone produced 54/60 valid MCQs, while the combined generator–validator–fallback pipeline delivered 57/60 (95%) valid items to learners. These results demonstrate that the tutor consistently adapts difficulty, safeguards quality, and provides explanatory feedback, creating a robust platform for interactive programming practice.

Contents

1. Introduction	6
1.1 Background & Motivation	6
1.2 Problem Statement	6
1.3 Aim	6
1.4 Objectives	6
2. Background & Related Work	7
2.1 Intelligent Tutoring Systems (ITS) & Adaptive Learning	7
2.2 MCQ Design & Validation	7
2.3 Large Language Models (LLMs) in Education	7
2.4 Emerging LLM-Powered Advisors and Pop-Quiz Tools	8
2.5 Adaptive Control & UX Considerations	8
2.6 Summary of Gaps & How This Project Addresses Them	8
3. Requirements	9
3.1 Essential Requirements	9
3.2 Recommended Requirements	9
3.3 Optional Requirements	9
4. Technical Specification — Tools & Technologies:	10
4.1 Frontend	10
4.2 Backend	11
4.3 Database	11
4.4 AI Integration	11
5. Development Methodology	12
5.1 How MVP is Implemented:	12
5.2 Why MVP + agile worked here:	12
5.3 Testing posture:	12
6. System Design & Architecture	13
6.1 High-Level Architecture	13
6.2 Data Model	14
6.3 Adaptive Engine	16
6.4 Prefetch contract (latency and correctness):	18
6.5 Fetcher, Generation, Validation & Fallback Pipeline:	18
7. System Implementation:	19
7.1 Codebase Layout	19
7.2 Data Ingestion	19

7.3 Prompt Templates & Output Schema	20
7.4 Generator Implementation	21
7.5 Validator Implementation	22
7.6 Adaptive Engine Methods	23
7.7 Frontend (Templates & UX)	24
7.8 Flask Endpoints & Flow	27
7.8.2 Session Setup	28
7.9 Persistence Layer	30
7.10 Security	34
8. Challenges	34
8.1 Creating & Labelling Datasets	34
8.2 GPT Hallucinations	34
8.3 User Experience & Latency Between Questions	35
8.4 Testing a Non-Deterministic, Data-Driven System	35
9. Testing & Evaluation	35
9.1 Objectives and Approach	35
9.2 Core Quiz Engine (Fetcher, Generator, Validator)	35
9.3 Endpoint and Integration Testing	36
9.4 Takeaways	36
10. Conclusion	37
11. Future Work	38
11.1 Content and Coverage	38
11.2 User Experience and Accessibility	38
11.3 Administrative Tools	38

1. Introduction

1.1 Background & Motivation

Adaptive, feedback-rich assessment is a long-standing goal in computing education. Traditional multiple-choice question (MCQ) platforms typically rely on static banks of items that date quickly, fail to reflect a learner’s evolving ability, and rarely explain *why* an answer is right or wrong. In contrast, contemporary AI systems can synthesize fresh items and provide tailored feedback at the pace of interaction. This project—**An AI Multiple-Choice Programming Exercises Tutor**—leverages that opportunity by combining a curated seed dataset with large language models (LLMs) to generate, validate, and deliver programming MCQs that adapt in real time to a learner’s performance. The result is a tutor that keeps challenge within the learner’s “productive struggle” zone, while offering concise, targeted explanations that support durable understanding. This work extends the preliminary plan you produced earlier and formalizes the final system and its UX refinements.

1.2 Problem Statement

Fixed MCQ repositories struggle to serve diverse cohorts: questions can be mis-levelled, memorized, or misaligned to an individual’s current knowledge. Moreover, item quality varies—distractors may be weak or ambiguous—and most systems provide limited feedback beyond “correct/incorrect”. The project addresses three concrete gaps:

1. **Personalisation:** deliver questions that match the learner’s present ability and adjust difficulty promptly after each response.
2. **Quality assurance for AI-authored items:** ensure generated MCQs are self-consistent and unambiguous before serving them.
3. **Timely feedback:** provide short, context-aware explanations that correct misconceptions or extend correct reasoning without interrupting flow.

1.3 Aim

To design, implement, and evaluate an AI-assisted programming MCQ tutor that **generates** and **validates** questions on demand, **adapts** difficulty per learner performance, and **explains** answers succinctly—all within a lightweight, secure web application.

1.4 Objectives

Core functional objectives

- Generate programming MCQs with GPT using a high-quality, labelled seed dataset (Python, Java, C) as exemplars.
- Implement an adaptive engine that follows a **two-step cycle per example**: generate a *same-level* question, then—conditioned on the learner’s response—generate a *higher* or *lower* one from the **same example**.
- Validate each generated item with an independent checker and **fallback** to the static example if validation fails.
- Provide concise, answer-aware explanations (“Know more”) after each submission.

Supporting objectives

- Offer secure authentication (hashed passwords) plus a **knowledge-based** reset flow (“remember word” with position checks) and guest access.
- Persist sessions and attempts for summaries and dashboards using a lightweight relational store (SQLite in development; deployable to MySQL/PostgreSQL).
- Present a clean, responsive UI for setup, quiz, feedback, results, and dashboards.

Non-functional objectives

- Keep *time-to-next question* low via background prefetch; maintain correctness and consistency under load.
- Externalise secrets (API keys, session secret) and follow minimal-privilege practices.
- Ensure repeatability (avoid duplicate examples within a session) and robustness (graceful fallbacks on API errors).

2. Background & Related Work

2.1 Intelligent Tutoring Systems (ITS) & Adaptive Learning

ITS traditionally combine a **domain model**, **learner model**, **pedagogical model**, and **interface** to deliver problems and feedback tailored to a student’s current state. Decades of work show that well-designed tutoring approaches can approach the effectiveness of one-to-one human tutoring (the “2-sigma” aspiration). Your project follows this lineage with a compact, transparent state machine that adjusts difficulty locally after each response.[15][20]

A closely related tradition is **Computerized Adaptive Testing (CAT)**: selecting each next item to keep difficulty near a learner’s ability, historically via Item Response Theory and later Bayesian methods. While your system uses lightweight heuristics (same → higher/lower), it serves the same goal of maintaining challenge without overload and can be extended toward CAT[17] if future work demands it. [21]

2.2 MCQ Design & Validation

High-quality MCQs hinge on: (i) a clear stem targeting a single objective, (ii) **plausible distractors** that reflect real misconceptions, and (iii) **exactly one best answer**. Authoritative guidance (e.g., Haladyna–Downing–Rodriguez; NBME’s Item-Writing Guide) emphasizes eliminating tricky wording, cueing, and implausible distractors, and auditing items for ambiguity. Programming MCQs add extra care for code formatting/versioning and edge cases. Your pipeline operationalizes these principles by **validating LLM-generated items** with an independent check and **falling back** to a vetted example if validation fails.[22]

2.3 Large Language Models (LLMs) in Education

LLMs enable two capabilities that matter for tutoring: **on-demand content synthesis** (e.g., fresh MCQs) and **contextual feedback** tied to a learner’s specific attempt. GPT-4 demonstrates strong performance on academic benchmarks[16], but variability and “hallucinations” necessitate guardrails. Recent studies across domains (computing, medical education, NLP courses) show that LLMs can produce **usable MCQs** and feedback, yet quality is uneven and often benefits from human or automated validation—exactly the role your validator plays.[23]

2.4 Emerging LLM-Powered Advisors and Pop-Quiz Tools

Recent research has increasingly examined the use of large language models (LLMs) for multiple-choice question (MCQ) generation in programming education. Comparative studies report that GPT-4 can produce coherent, single-correct-answer MCQs aligned to learning objectives, with quality approaching that of human-crafted items. Pilot systems have also explored adaptive quiz delivery, using retrieval-augmented generation (RAG) to personalise engineering content dynamically.

Beyond isolated quiz generation, some advanced platforms (e.g., Iris) integrate GPT-based tutoring into broader programming environments, providing contextual assistance during code authoring. Other research prototypes have trialled LLM-generated quizzes with adaptive scaffolding in block-based programming contexts, or hybrid approaches combining LLM content creation with explicit student modelling.

Against this backdrop, the present system distinguishes itself by combining three specific design choices:

- **On-demand MCQ generation**, tailored to programming and multilingual contexts;
- **Binary validation for correctness**, enforcing the “exactly one defensible answer” rule; and
- **An adaptive, user-driven difficulty engine**, running entirely server-side with no client-side dependencies.

This blend of novelty and guardrails makes the tutor both innovative and practical, striking a balance between leveraging generative AI and maintaining reliability for learners.

2.5 Adaptive Control & UX Considerations

Adaptation works only if interaction retains **rhythm** (fast turn-taking, immediate feedback) and preserves learner autonomy. Research on **formative feedback** underscores that timely, targeted feedback improves learning when it clarifies misconceptions without derailing engagement—hence your short, answer-aware explanations and the **permissioned difficulty change** (modal). Your concurrency design (background prefetch + *display-only Next*) reduces wait time without duplicating work, supporting continuity of focus.

2.6 Summary of Gaps & How This Project Addresses Them

Gap in the literature/practice	Consequence in typical MCQ systems	This project’s approach
Static item banks don’t match individual ability	Too easy/hard; poor engagement	Per-response adaptation: same-level first, then higher/lower from the same example
Inconsistent quality of AI-generated items	Ambiguity; multiple “correct” options	Validator gate independently confirms a single correct option; fallback to vetted example
Feedback is generic or delayed	Misconceptions persist; low retention	Answer-aware explanations (“Know more”) immediately after each response

Latency between questions	Breaks flow; disengagement	Background prefetch while reading feedback; Next waits/consumes (single-flight)
Hard jumps in difficulty	Loss of control; frustration	User-permissioned difficulty change (modal) when performance thresholds are met

3. Requirements

The system requirements were classified into three categories: **essential**, **recommended**, and **optional**. Essential requirements define the core functionality necessary for an operational adaptive quiz system. Recommended requirements enhance usability and engagement, while optional requirements represent advanced features that can extend the system's academic and pedagogical value.

3.1 Essential Requirements

These represent the minimum capabilities required for the system to deliver programming quizzes with adaptive behaviour and AI support. All essential requirements together, they define the baseline functionality of the system.

Id	Requirement	Description
E1	MCQ Presentation	Display multiple-choice questions and options to the learner in a clear format.
E2	Answer Checking	Validate user-submitted answers and compute scores accordingly.
E3	Topic Selection	Allow learners to choose programming languages (Python, Java, C).
E4	Question Generation	Use the OpenAI API to generate new MCQs dynamically.
E5	AI Explanation	Provide AI-generated explanations tailored to the learner's answer.
E6	Adaptive Difficulty	Modify question difficulty in real time based on learner performance.

3.2 Recommended Requirements

Recommended requirements improve the user experience and provide meaningful personalisation beyond the baseline. These features provide persistence, continuity, and greater engagement for learners.

Id	Requirement	Description
R1	Web Interface	Flask-based frontend providing interactive, server-rendered pages.
R2	User Profiles	Persist user accounts, track scores, and maintain progress across sessions.
R3	Log Incorrect	Store incorrect responses for later review and analysis.

3.3 Optional Requirements

Optional features extend the educational impact of the system and provide opportunities for enhanced engagement and motivation.

Id	Requirement	Description
O1	Reward System	Assign points and bonuses for correct answers to increase learner motivation.
O2	Personalized Dashboard	Provide a dynamic, personalised dashboard with visualisations of progress.
O3	“Know More” Explanations	Allow learners to request deeper, tailored explanations after viewing feedback.

Most optional features have also been realised. The **Reward System** is implemented through points and streaks, while the **Personalised Dashboard** visualises accuracy, streaks, and performance trends across sessions. Instead of a full chatbot (initially proposed), a **“Know More” button** was added. This feature partially fulfils the same purpose by allowing learners to request richer, AI-generated explanations contextualised to their answers, thereby supporting deeper understanding without introducing the complexity of an interactive conversational agent

4. Technical Specification — Tools & Technologies:

Layer	Technology	Purpose / Notes
Frontend	HTML5, CSS, Vanilla JS, Jinja2 templates	Pages: login, signup, forgot password, home, quiz, result, dashboard, session detail
Charts	Chart.js (CDN)	Topic accuracy; accuracy by difficulty on the dashboard
Backend	Python 3.1, Flask	Templating, routing, sessions, request handling
Concurrency	threading (Python stdlib)	Background prefetch of the next MCQ; Event for single-flight.
AI SDK	OpenAI Python SDK (OpenAI client)	MCQ generation, validation, “Know more” explanations
Data store	SQLite (dev)	Tables: users, quiz_sessions, quiz_attempts
Config & secrets	python-dotenv , OS env vars	OPENAI_API_KEY, FLASK_SECRET_KEY, etc.

4.1 Frontend

HTML5, CSS3 are the languages used for the development of entire User Interface (UI). The interface is intentionally built with **pure HTML and CSS** [2], without much relevance on client-side JavaScript. Entire frontend shares same style.css and theme.css files which were written in CSS3. For this application, these choice delivers three concrete benefits:

- Speed and reliability.** Server-rendered pages reach first paint quickly and behave consistently across browsers. There is no JavaScript bundle to download or execute, so the quiz and feedback cycles feel instant—especially important as learners move rapidly between questions.
- Security and simplicity.** With no client logic, **no secrets** or API calls are exposed in the browser. All sensitive operations (authentication, AI generation, validation) remain server-side. This also reduces the attack surface and eliminates common SPA concerns.
- Consistent UX across views.** Shared CSS3 files provides a unified layout system for login, signup, password reset, home, quiz, results, dashboard, and session detail pages. Shared

styles guarantee alignment, spacing, and code-block readability (crucial for programming MCQs) without the complexity of a front-end framework.

4.2 Backend

The backend of the system is built in **Python 3.1**, using the **Flask** micro web framework [1] as its foundation. This choice reflects the need for a lightweight, composable architecture that can clearly express the system's core responsibilities without unnecessary overhead. Supporting libraries include **flask_mysqldb** [10] for database connectivity, **werkzeug.security** [3] for secure credential handling, and **openai** for AI-driven question generation, validation, and explanations.

Flask's routing model maps cleanly to the quiz workflow. Flask's built-in session handling, backed by signed cookies, provides just enough state to maintain flow between requests while avoiding the complexity of heavier frameworks. Security is addressed through **werkzeug.security**, which provides standard-compliant password hashing and verification functions.

For persistence, **flask_mysqldb** integrates MySQL with minimal boilerplate [12]. Connections are opened per request, and parameterised queries are executed through cursors. This direct approach simplifies reasoning about state changes in the database, particularly for writing quiz sessions and attempts.

Overall, Flask's minimalism proves advantageous. By offering only what is necessary—routing, session management, and integration hooks—it allows the project to focus on its distinctive features: the adaptive engine and AI pipeline. Even concurrency features, such as background prefetching of the next question, integrate neatly without altering the web contract. The result is a backend that is lean, secure, and purpose-built for adaptive tutoring.

4.3 Database

The main purpose of database in this system is to maintain persistent, ACID-compliant storage for **user accounts**, **authentication details**, **quiz sessions**, **per-question answers**, and derived **performance metrics**. MySQL 9.1. fulfil the requirements and offers scalability for future. This enables reliable resumption, auditability, and robust dashboards. Core tables are **users**, **quiz_sessions**, and **quiz_attempts**. [4][12]

4.4 AI Integration

AI is the core of this application so it important that the model to be highly reliable and time efficient. Hence, this application integrates the OpenAI API [8] to provide the core intelligence behind question generation, validation, and adaptive feedback. This integration transforms a static multiple-choice system into a dynamic tutoring environment.

At its foundation, the system employs the API for **generation** of new multiple-choice questions. These questions are conditioned on curated examples from the dataset, ensuring they are stylistically aligned with vetted material while targeting the requested difficulty level (same, easier, or harder).

To maintain reliability, every generated question is passed through a **validation layer**. In this step, the model independently re-derives the correct option to confirm that exactly one defensible answer exists. If the validator identifies inconsistencies or uncertainty, the system automatically falls back to the curated example. This safeguard allows the tutor to provide on-demand variety without compromising correctness, which is critical in an educational context.

Beyond question creation, the API is also used for **explanations and enrichment**. After each attempt, the system delivers concise, tailored feedback that either addresses misconceptions when the answer is wrong or deepens understanding when the answer is correct. Learners can also choose to request

additional detail through the “Learn More” feature, which provides an expanded, attempt-aware explanation that links the concept to practical examples.

Finally, the integration is implemented with operational safety in mind. All API calls are made server-side, with keys stored securely in environment variables. Failures such as timeouts or validation errors result in smooth fallbacks to static questions, ensuring that the learner’s experience is never disrupted. In this way, the OpenAI API provides the “brains” of the tutor, while the validator and fallback act as “guardrails,” striking a balance between innovation and reliability.

5. Development Methodology

This project was delivered using an **Minimum Viable Product-first (MVP), agile** approach: build the smallest end-to-end tutor that proves value, then iterate in short cycles to improve reliability (validator/fallback), adaptivity (state machine), and user experience (prefetch + permissioned difficulty change). The guiding strategy was to prioritise **core functionality** and, adding refinements only after the core loop was robust.

5.1 How MVP is Implemented:

Work largely followed the preliminary schedule. **Weeks 1–2** focused on a brief literature review, learning AI/LLM concepts and sketching the architecture, followed by a **CLI prototype**. Firstly created a High quality, topic and difficulty labelled dataset with 100+ MCQs per language (C/Java/Python) in JSON format. Build a **fetcher.py** module to load the dataset of user requested language and fetch MCQs by applying difficulty filters.

In the next cycle started with basic **AI integration** and development of **generator.py** module. Fetched MCQ is used fed to generator to generate a curated MCQ using GPT-4.0 [8]. To address the inaccurate responses from Generator caused by AI hallucinations a **validator** module is built to double check for any inconsistencies. These Three modules act as the core MCQ engines. After that implemented the adaptive difficulty logic as a compact state machine and integrated with the core modules. Then **Flask web interface** replaced the CLI, keeping server-side rendering to avoid exposing secrets. With this a minimum viable product is confirmed and later on started working on **UI/UX enhancements, user accounts and session tracking**, then optional items.

5.2 Why MVP + agile worked here:

The core loop (fetch → generate → validate → evaluate user) is where most of the risk sits. Building it early gave rapid feedback on where to put guardrails (validator-gate, fallbacks) and what to postpone. Short iterations made it easy to slot in UX improvements (e.g., permissioned difficulty changes via a modal) without destabilising the loop.

5.3 Testing posture:

Unit checks were carried for every module individually after along with the development process. Moved to the next module only after confirming the current module is reliable. This is because every module is dependant to the next. Integration testing is also covered while coupling the modules for example (fetcher + generator + validator). So testing is carried out throughout the development lifecycle for every iteration. We will see more details and metrics about testing in coming chapters.

6. System Design & Architecture

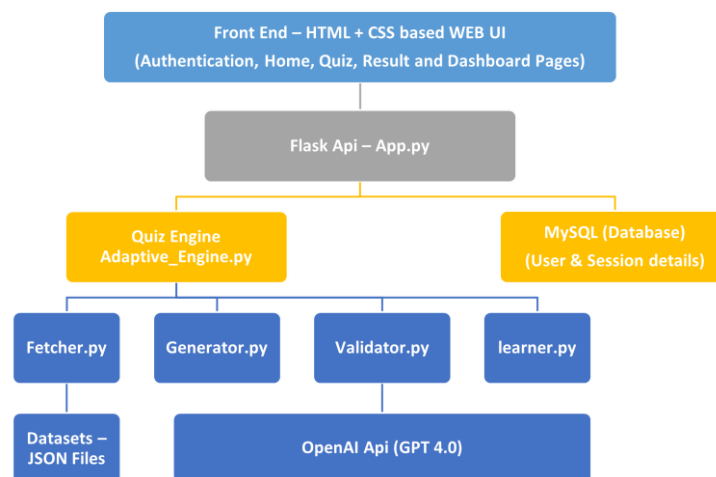
This section describes the system's structure, the flow of data and control. Also explains adaptive logic, and the concurrency model that underpins the “fast next question” experience. It also sets out security, scalability, and observability considerations to support a production deployment.

6.1 High-Level Architecture

The system is a lightweight web interface that serves programming MCQs, adapts difficulty per response, validates AI-generated items, and explains answers concisely—all while keeping latency low via background prefetch.

- **Web/UI (Flask + Jinja2):** login/signup/reset, home (setup), quiz loop, difficulty-change modal, results, dashboard, session details.[2]
- **Adaptive Engine (per session):** The state machine; tracks performance & streaks; proposes ± 1 difficulty **only at** user-permission. Coordinates **background prefetch** and expose an Event so **Next is display-only** (waits & consumes).
- **Content Services:**
 - **Fetcher** loads the corresponding dataset and selects an **example (E)** at the current difficulty (without repeats).
 - **Generator** creates a MCQ (**S**) of same difficulty level as (**E**) and the second MCQ(**C**) is (**conditional based on user performance of (S), higher/lower**) from the **same (E)**.
 - **Validator** confirms the generated MCQs (S/C) are valid and correct; on failure it **fallback to (E)**.
 - **Learner** produces more detailed explanations upon User request.
- **Persistence (MySQL):** Stores User and session data in “users”, “quiz_sessions”, “quiz_attempts” for authentication, per-run aggregates, and per-attempt analytics.
- **Seed datasets (JSON):** 100 MCQs per language (Python/Java/C) with topic & difficulty labels.
- **OpenAI API (cloud):** used by generator/validator/learner; calls are server-side only.

Architecture diagram:



(Figure 1, Architecture Overview)

6.2 Data Model

Below is the data model used by the tutor. It captures authentication, per-session aggregates, and per-question attempts. I've also given you a clean ER diagram you can paste into the report.

6.2.1 Entities and Relationships

Users: Holds registered user accounts and the authoritative identity record for logged-in players.

Field	Type	Null	Key	Default	Extra
id	int	NO	PRI	NULL	auto_increment
username	varchar(50)	NO	UNI	NULL	
password	varchar(255)	NO		NULL	
remember_word	varchar(100)	NO		NULL	
created_at	timestamp	YES		CURRENT_TIMESTAMP	DEFAULT_GENERATED

quiz_sessions: summarizes a single quiz run—who played, when, the difficulty arc, and the final scores/streaks.

Field	Type	Null	Key	Default	Extra
Id	bigint	NO	PRI	NULL	auto_increment
user_id	int	YES	MUL	NULL	
Username	varchar(50)	NO		NULL	
Language	varchar(20)	NO		NULL	
start_difficulty	tinyint	NO		NULL	
final_difficulty	tinyint	NO		NULL	
total_questions	int	NO		NULL	
Correct	int	NO		NULL	
Bonus	int	NO		NULL	
final_score	int	NO		NULL	
started_at	timestamp	NO		CURRENT_TIMESTAMP	DEFAULT_GENERATED
ended_at	timestamp	YES		NULL	
longest_streak	int	NO		0	
five_streaks	int	NO		0	

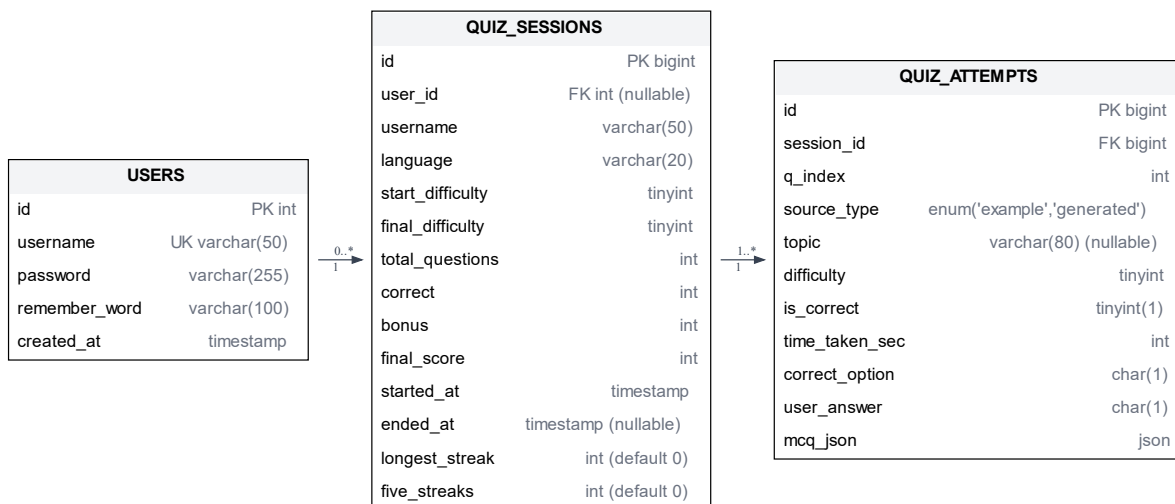
quiz_attempts: the granular “question-level” log used to compute accuracy, timing, and per-topic stats.

Field	Type	Null	Key	Default	Extra
Id	bigint	NO	PRI	NULL	auto_increment
session_id	bigint	NO	MUL	NULL	
q_index	int	NO		NULL	
source_type	enum('example','generated')	NO		NULL	
Topic	varchar(80)	YES	MUL	NULL	
Difficulty	tinyint	NO		NULL	
is_correct	tinyint(1)	NO		NULL	
time_taken_sec	int	NO		NULL	
correct_option	char(1)	NO		NULL	
user_answer	char(1)	NO		NULL	
mcq_json	json	NO		NULL	

6.2.2 How the engine populates the model

- On **/start**, a quiz_sessions row is inserted with the chosen language, starting difficulty, and question count; started_at is set.
- On each **/quiz** submission, one quiz_attempts row is written with timing, topic, difficulty, correctness, and whether the served question was AI or example. Aggregates (score, streaks, correct_answers, bonus_points) are updated in quiz_sessions.
- On **/result**, ended_at is set and final_difficulty is recorded.

6.2.3 ER Diagram



(Figure 2, Entity Relationship)

6.2.4 MCQ Object Contract (in-memory / JSON)

This schema is shared between the dataset and generated items:

```

{
  "id": "string",
  "topic": "string",
  "difficulty": 1,
  "question": "string",
  "code": "string",
  "options": { "A": "...", "B": "...", "C": "...", "D": "..." },
  "correct_option": "A|B|C|D",
  "explanation": "string"
}
    
```

Here are the few benefits of this JSON format:

Consistent, Machine-Friendly Format (JSON)

- The standardized JSON structure makes it easy to:
 - Ingest into databases or vector stores
 - Feed into language models or embedding systems
 - Use with APIs or LLM pipelines (e.g. ChatGPT, LangChain, RAG apps)

Difficulty-Based Segmentation

- Difficulty levels (1–5) support:
 - Useful for **search, filtering, tagging**, or feedback loops
 - **Adaptive testing** (adjusts question difficulty as the user performs)
 - **Personalized learning paths** (easy → hard)
 - Curriculum design, pacing, and challenge control

Topic Tagging Enables Targeted Learning

- Each question has a topic tag (Functions, Inheritance, Lambdas, etc.)
 - Supports **topic-specific drills** and dynamic question generation

Ready for LLM Alignment

- This is also future proof, for:
 - Build datasets for **supervised fine-tuning** (SFT)
 - Construct **evaluation benchmarks** for LLMs on reasoning
 - Power **RAG pipelines** (Retrieval-Augmented Generation) where the model retrieves explanations or similar questions

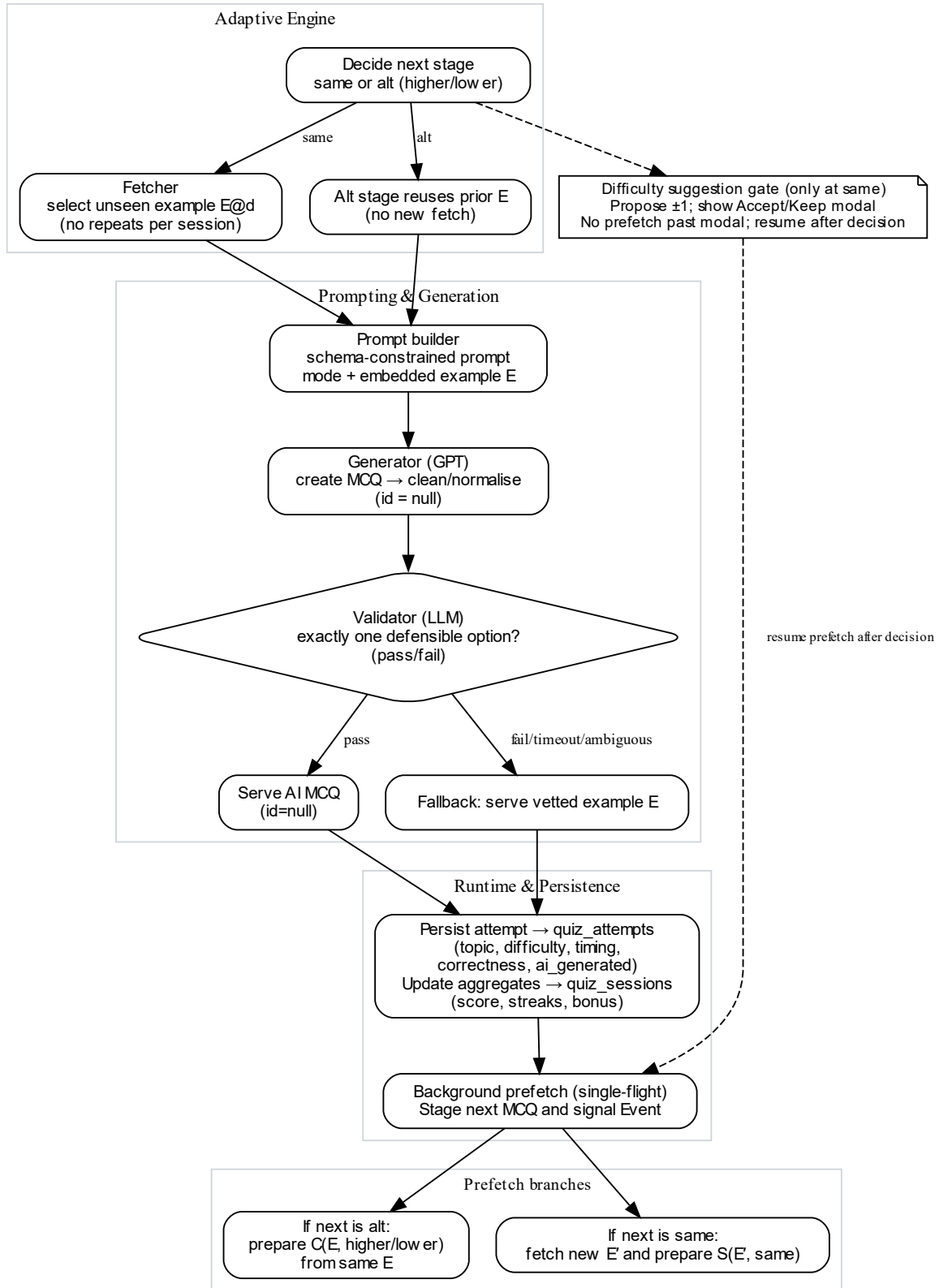
6.3 Adaptive Engine

The adaptive engine is a compact, transparent state machine which is nothing but a adaptive difficulty logic written in python that sequences questions per example, handles performance metrics, proposes difficulty shifts (with user permission), and coordinates background prefetch so the quiz feels instant. This engine also acts as a bridge between the fetcher, generator and validator. These concepts are explained in detail in the following contexts.

6.3.1 State machine (overview)

This state machine handles the OpenAI api calls [6] and chooses prompt based on the **Stage** the user is in. Basically, this system has two stages called “**same**” and “**alt**”. In the **same** stage the generator is prompted to synthesize an MCQ (S) with same level difficulty as the example MCQ (E). Whereas in the **alt** stage the prompt is conditional if, the user answers (S) correct then the generator is prompted to generate a more complex or difficult MCQ, else lower. After the conditional MCQ (C) is answered, the machine return to stage **same** and fetch a new example for the next iteration. So, this cycle continues till the user requested number of MCQs are generated.

So, each example (E) serves two MCQs, one same-level(S) and one conditional (C). The point to note is that **C** is always derived from the same **E** as **S**. If there is any failure in generation of the MCQ then system fallbacks to **E**. And another case for fallback is if, the validator identifies any inconsistencies in the generated MCQs (**S/C**) and returns False.



(Figure 3, Flow of Adaptive_Engine)

6.3.2 Performance counters & difficulty proposals

The State machine is also responsible for handling the user's performance metrics and streaks. These metrics are used for change in difficulty proposals. System proposes ± 1 difficulty level, **Suggested up** when sustained performance is high (e.g., `user_performance` ≥ 6), **down** when `wrong_streak` ≥ 3 . This proposal was shown via a modal and applied only after user Accept/deny. A small **cooldown** prevents back-to-back proposals, accepting resets counters appropriately. These metrics are not only useful for adaptive logic but also contributes for User Experience(UX) because metrics like `longest_correct_streak`, and `five_streaks` are used to display in the dashboard which keep up the motivation.

6.4 Prefetch contract (latency and correctness):

To keep the interaction snappy, the engine **prefetches** the next MCQ in the background while the learner is reading feedback. Prefetching is *single-flight*: at most one background job is allowed at a time, and the *Next* action never creates a second generator.

It starts immediately after a submission (POST /quiz). If the next stage is *alt*, the engine builds C (higher/lower) from the same E used for S. If the next stage cycles back to same, it fetches a new example E' and prepares S(E', same) at the current difficulty. And in the case of change in difficulty proposal, the engine does not prefetch past the proposal. Instead, it signals readiness so Next shows the Accept / Keep modal immediately.

This shows a significant impact on the UX. The learner sees the next question appears instantly and short "preparing next question..." state only if the generation finishes after *Next* is clicked. So most of the times the User interaction will be snappy.

6.5 Fetcher, Generation, Validation & Fallback Pipeline:

6.5.1 Fetcher

The question pipeline begins with the **adaptive engine** deciding what the next step should be: a *same-level* item or a *conditional* item (higher if the previous response was correct, lower if it was not). That choice determines both the example used and the prompt sent to the generator. When the engine is at the *same* stage, it asks the **fetcher** to supply an unseen **example MCQ (E)** at the current difficulty, drawn from the curated JSON datasets for Python, Java, and C. The fetcher enforces a simple hygiene rule—no repeating example IDs within a session—so that variety and topic coverage remain high. When the engine is at the *alt* stage, no new example is fetched: the conditional item must be derived from the **same E** that seeded the previous question to preserve topical coherence.

6.5.2 Generator

With E in hand, the engine assembles a **schema-constrained prompt** that specifies the desired mode (*same*, *higher*, or *lower*), embeds salient details of the example, and requires a strict output structure: a single stem, an optional code block, exactly four options (A–D), an explicit `correct_option`, and a short explanation. The **generator** (GPT) returns a candidate MCQ which is then lightly cleaned—options are normalised, stray prose is trimmed, fenced code is moved into the code field—and its id is set to null so later analytics can distinguish AI-authored items from static examples.

6.5.3 Validator

Every generated MCQ object then passes through an **independent validator**. This second LLM call re-derives the correct option from the stem and code, checking that exactly one option is defensible and that the distractors are not co-correct. The validator's decision is deliberately binary: *pass* or *fail*. On a **pass**, the AI item is accepted as-is and becomes the question the learner sees. On a **fail**—including malformed output, ambiguity, or any model/SDK error[8]—the system **falls back** to the vetted example E for that step. This *generate* \rightarrow *validate* \rightarrow *fallback* contract guarantees that the learner

never encounters a broken or ambiguous question while still benefiting from fresh AI content whenever it passes quality checks.

7. System Implementation:

7.1 Codebase Layout

The project is organised as a small, readable Flask application with clearly separated modules for fetching examples, generating/validating MCQs, adaptivity, and explanations. Static datasets live under `data/`, and server-rendered templates live under `templates/`. Below is the detailed file structure of the project.

```
ai_mcq_tutor_project/
├─ app.py                # Flask entrypoint: routes, sessions, view rendering
├─ adaptive_engine.py    # Per-session state machine (same → alt), counters, prefetch
├─ fetcher.py           # Loads curated example MCQs from /data, avoids repeats
├─ generator.py         # OpenAI (classic `openai`) - generate S/C items, normalize
├─ validator.py        # OpenAI (classic `openai`) - independent answer check
├─ learner.py          # OpenAI (new `OpenAI` client) - "Know more" explanations
├─ prompt_templates.py  # Prompt builders for same / higher / lower, strict schema
├─
├─ data/
│   ├─ python.json      # 100 curated MCQs (topic + difficulty labels)
│   ├─ java.json
│   └─ c.json
├─
├─ templates/          # Jinja2 server-rendered pages (HTML)
│   ├─ login.html
│   ├─ signup.html
│   ├─ forgot_password.html
│   ├─ forgot_password_step2.html
│   ├─ home.html
│   ├─ quiz.html        # Shows item; feedback block; difficulty modal as needed
│   ├─ result.html
│   ├─ dashboard.html   # Charts (topic accuracy, by difficulty)
│   └─ session_detail.html
├─
├─ static/             # single CSS3 layout/styles.css
│   └─ styles.css
├─
├─ requirements.txt    # Flask, flask_mysql, openai, python-dotenv, etc.
├─ .env               # OPENAI_API_KEY, FLASK_SECRET_KEY, MySQL creds (env-only)
└─ README.md          # Setup, run, and deployment notes
```

(Figure 4, Project File Structure)

7.2 Data Ingestion

The ingestion layer is responsible for transforming curated JSON files into reliable example items that seed question generation. Its goal is to provide a consistent schema, fast access by language and difficulty, and a guarantee that users do not encounter repeated examples within the same session. This ensures that the adaptive engine always begins with a vetted baseline before invoking AI generation.

The system's data sources are the JSON files stored under `data/` (e.g., `python.json`, `java.json`, `c.json`). Each record in these files conforms to a uniform schema, including an identifier, topic, difficulty rating (1–5), question text, optional code snippet, four answer options, the correct option, and an explanatory note. These datasets act as the ground truth for topic and difficulty labels. They also provide the fallback path whenever AI generation fails validation or becomes unavailable, ensuring continuity in the learner's experience. Below is an example of a data object from `python.json`.

```
{
  "id": "PY2-STR-028",
  "topic": "Strings",
  "difficulty": 2,
  "question": "What will be the output of this code?",
  "code": "s = \"Python\"\nprint(len(s))",
  "options": {
    "A": "6",
    "B": "5",
    "C": "7",
    "D": "Error"
  },
  "correct_option": "A",
  "explanation": "The string 'Python' has 6 characters, so 'len(s)' returns 6."
},
```

(Figure 5, JSON Object Example)

On session start, the fetcher loads the relevant JSON file into memory according to the user's chosen language. The dataset is then filtered by the requested difficulty level, and items are randomly sampled to avoid predictable patterns. Storing the dataset in RAM eliminates disk I/O during the quiz loop, making retrieval effectively instantaneous and supporting efficient background prefetching.

7.3 Prompt Templates & Output Schema

A central requirement of the system is that the language model produce outputs that are **predictable**, **creative** and **checkable**. To achieve this, prompt templates are designed to clearly instruct the model on what to generate and how to format it, while the output schema (together with the validator) ensures that each item is structurally sound, unambiguous, and safe to serve to learners.

Prompt design revolves around a labelled example multiple-choice question (E) combined with an explicit **mode**: *same*, *higher*, or *lower*.

- In **same mode**, the model synthesises a new question that targets the same concept and difficulty as E without copying its stem or code verbatim.
- In **higher mode**, the model increases cognitive load relative to E—for example, by combining related ideas, introducing subtle runtime behaviour, or requiring reasoning across multiple lines of code.
- In **lower mode**, the model simplifies the task by reducing interactions, shortening the stem, or focusing on recognition of a single construct rather than reasoning across several.

All templates also embed a set of **shared instructions** that apply regardless of mode. These constraints enforce consistency and quality. Together, the templates and schema provide the necessary **guardrails** for consistent and checkable LLM output. The model has room to generate fresh, varied items, but within boundaries that ensure pedagogical value and structural validity. This design balances creativity with reliability, enabling the system to expand its coverage adaptively without sacrificing trustworthiness.

Example MCQ	Generated MCQ
{ 'id': 'PY1-DTV-007', 'topic': 'Data Types & Variables', 'difficulty': 1,	{ 'question': 'Which of the following expressions will correctly check if a variable `var` is truthy in Python?',

<pre> 'question': 'What type of value is returned by the `input()` function?', 'code': "", 'options': { 'A': 'int', 'B': 'float', 'C': 'str', 'D': 'bool' }, 'correct_option': 'C', 'explanation': 'By default, `input()` returns a string. It must be explicitly cast to another type if needed.' } </pre>	<pre> 'code': "var = 'Hello'", 'options': { 'A': 'if var:', 'B': 'if var == True:', 'C': 'if var is True:', 'D': 'if bool(var) == False:' }, 'correct_option': 'A', 'explanation': "In Python, using 'if var:' checks if the variable is truthful directly. The other options either use incorrect comparisons or negate the truthiness check." } </pre>
---	--

7.4 Generator Implementation

The generator's role is to transform a labelled example multiple-choice question (E) and a target mode (*same*, *higher*, or *lower*) into a schema-conformant item that is safe to validate and fast to render. It is implemented using the OpenAI Python SDK on the server side, ensuring that API keys remain secure and are never exposed in the browser[6].

```

if self.stage == "same":
    # Prepare new example E' and G1(E', "same")
    example = fetch_next_unique_mcq(self.language, self.current_difficulty, self.fetched_ids)
    prompt = build_user_prompt(example, "same")
    gen = generate_mcq(example, prompt)

else:
    # stage == "alt" → next is G2(E, higher/lower)
    prompt_type = "higher" if self.last_correct else "lower"
    prompt = build_user_prompt(self.last_example, prompt_type)
    gen = generate_mcq(self.last_example, prompt)

```

(Figure 6, Stage management in Adaptive Engine)

When the adaptive engine requests a new item, it passes the tuple (*E*, *prompt*) to the generator. The generator then invokes the Chat Completions API and builds a mode-aware MCQ. The expected response is a single JSON object containing the required fields: question, optional code, options (A–D), correct_option, explanation, topic, and difficulty.

```

def generate_mcq(example_mcq: dict, user_prompt: str) -> dict:
    print(example_mcq)
    try:
        response = openai.chat.completions.create(
            model="gpt-4o",
            temperature=0.8,
            max_tokens=400,
            messages=[
                {"role": "system", "content": SYSTEM_PROMPT},
                {"role": "user", "content": user_prompt}
            ],
            response_format={"type": "json_object"}
        )

        content = response.choices[0].message.content
        gen_q= json.loads(content) # ensures it's valid JSON
        print(gen_q)
        return clean_generated_mcq(gen_q)

```

(Figure 7, Code Snippet of Generator OpenAI Api Call)

Immediately after the API returns, a series of **post-processing and normalisation steps** are applied. The response is parsed and any extraneous text or code fencing is stripped, leaving only the JSON payload. The generator checks that all four options exist and are well-formed strings, removes trivial duplicates, and converts any markdown code fences into plain text in the code field. To distinguish generated items from curated examples, the id field is set to null. If any of these checks fail—because of malformed JSON, missing keys, or empty options the generator reports an error instead of producing a candidate. The adaptive engine then falls back to the vetted example E, guaranteeing continuity in the quiz loop.

The generator also respects explicit **difficulty steering**. It does not attempt to infer difficulty itself; instead, it follows the adaptive engine’s instructions embedded in the prompt. From an architectural perspective, the generator runs most often in the **background prefetch thread**, triggered immediately after a learner submits an answer.

Overall, the generator is deliberately constrained. By combining structured prompts with strict normalisation, it converts a creative model into a reliable component: either a well-formed MCQ is produced and validated, or the system instantly falls back to the curated dataset.

7.5 Validator Implementation

The validator serves as a deterministic gatekeeper that decides whether an AI-generated multiple-choice question (MCQ) is safe to present to learners. Its sole responsibility is to re-derive the correct answer from the given stem, code, and options, and then compare this to the declared answer. The output is deliberately binary: *True* if the item is valid, *False* if it should be rejected. This ensures the quiz loop never delivers an ambiguous or malformed question.

```
SYSTEM_PROMPT = """You are a programming MCQ validator.
You will receive a dict containing:
- A question with code
- Options (A, B, C, D)
- The marked correct_option
- An explanation

Your job is to:
1. Simulate or analyze the code logically
2. Determine the correct answer (A/B/C/D)
3. Check if the marked correct_option is the right one

If everything is correct, return 'True'
If anything is incorrect, return 'False'

Only return 'True' or 'False'
No explanation or extra comments
"""
```

(Figure 8, Validator system prompt)

The validator is implemented as a simple function `validate_mqc(mcq: dict)` which accepts the full candidate item, including the question stem, optional code, four options (A–D), the declared `correct_option`, and an explanation. Internally, it constructs a compact user prompt embedding the entire MCQ in JSON form and sends this to the Chat Completions API. A system instruction directs the model to analyze or “simulate” the code, determine the true correct answer, and return only a boolean verdict. [6]

```
response = openai.chat.completions.create(
    model="gpt-4o",
    temperature=0,
    max_tokens=400,
    messages=[
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": user_prompt}
    ]
)
```

(Figure 9, GPT Model Parameter for Validator)

The configuration fixes the model to *gpt-4o* with temperature=0, ensuring consistency and eliminating stochastic variation. The reply is parsed with a simple string check: if the response contains "TRUE" (case-insensitive), the function returns *True*; otherwise it returns *False*.

7.6 Adaptive Engine Methods

The adaptive engine is the central coordinator that decides what to ask next, how difficult it should be, and when to generate it. It encapsulates session state (difficulty, counters, streaks, stage), enforces the two-per-example rule, proposes difficulty adjustments, and manages background prefetching so that the "Next" button is always display-only—consuming a staged question rather than triggering generation. This design keeps the user experience smooth and prevents race conditions.

The MCQ synthesis tasks discussed in the previous chapters (7.4)(7.5) are also handled by the Adaptive engine. Along with those MCQ generation and validation tasks, it is also responsible for core loop logic, adaptive difficulty and calculating user performance metrics. The engine centralizes **adaptivity** and **timing** so the rest of the app stays simple. Few other core methods are discussed below.

The method **submit_answer(mcq, user_answer, time_taken)** records each attempt and updates the engine state. It persists a row in `quiz_attempts` with details such as topic, difficulty, correctness, timing, and whether the question was AI-generated, while also updating aggregates in `quiz_sessions`. At the in-memory level, it increments counters, adjusts **user_performance** and **wrong_streak**, and tracks streaks for gamification. "user_performance" is used as metric to propose increasing difficulty level. And it is calculated as if: user answers "same" difficulty level MCQ correct, "+1", and if user answers "high" difficulty also correct "+2", and decremented by "-1" for every 2 streak of wrong answers. If `wrong_streak` is greater than 3 the engine propose to lower the difficulty level.

```
# Adaptive counters (feed into next proposal)
if self.last_prompt_type == "same":
    if correct:
        self.user_performance += 1
        self.wrong_streak = 0
    else:
        self.wrong_streak += 1
elif self.last_prompt_type == "higher" and correct:
    self.user_performance += 2
elif self.last_prompt_type == "lower" and not correct:
    self.user_performance -= 1
    self.wrong_streak += 1
```

(Figure 10, User performance calculation)

Prefetching and staged consumption is implemented by importing the **Event** class from Python's **threading** module, the method `prefetch_next()` prepares the next question in a background thread. Its behaviour depends on the current stage:

- If `stage == "alt"`, the engine generates a conditional item C (higher or lower) derived from the last example E.
- If `stage == "same"`, it fetches a new example E' at the current difficulty, generates a same-level item, and validates it.

Difficulty proposals and decisions are actually handled by three methods

```
def adjust_difficulty(self):  
    """  
    Compute a *proposed* difficulty based on performance,  
    but do NOT apply it here.  
    """  
    # Thresholds based on your earlier logic  
    if self.user_performance >= 7 and self.current_difficulty < 5:  
        return self.current_difficulty + 1  
    elif self.wrong_streak >= 3 and self.current_difficulty > 1:  
        return self.current_difficulty - 1  
    return self.current_difficulty
```

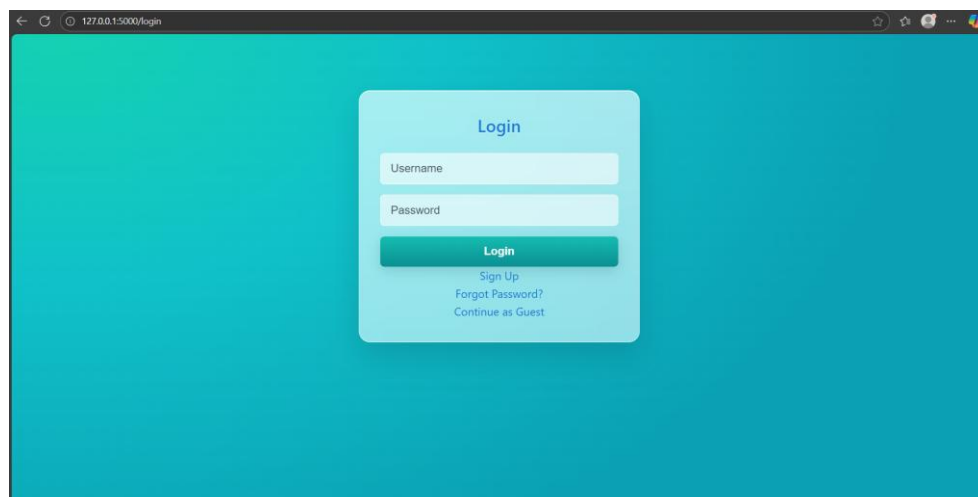
(Figure 11, Difficulty adjustment)

- **adjust_difficulty()** method checks the `user_performance` and `wrong_streak` metrics and compute the proposed difficulty method based on user performance.
- **propose_difficulty(self)**: Set a suggestion if conditions are met, respecting cooldown.
- **apply_difficulty_decision(self, accept: bool)**: Called by `/quiz/decide` after user clicks Accept or Keep Current.

7.7 Frontend (Templates & UX)

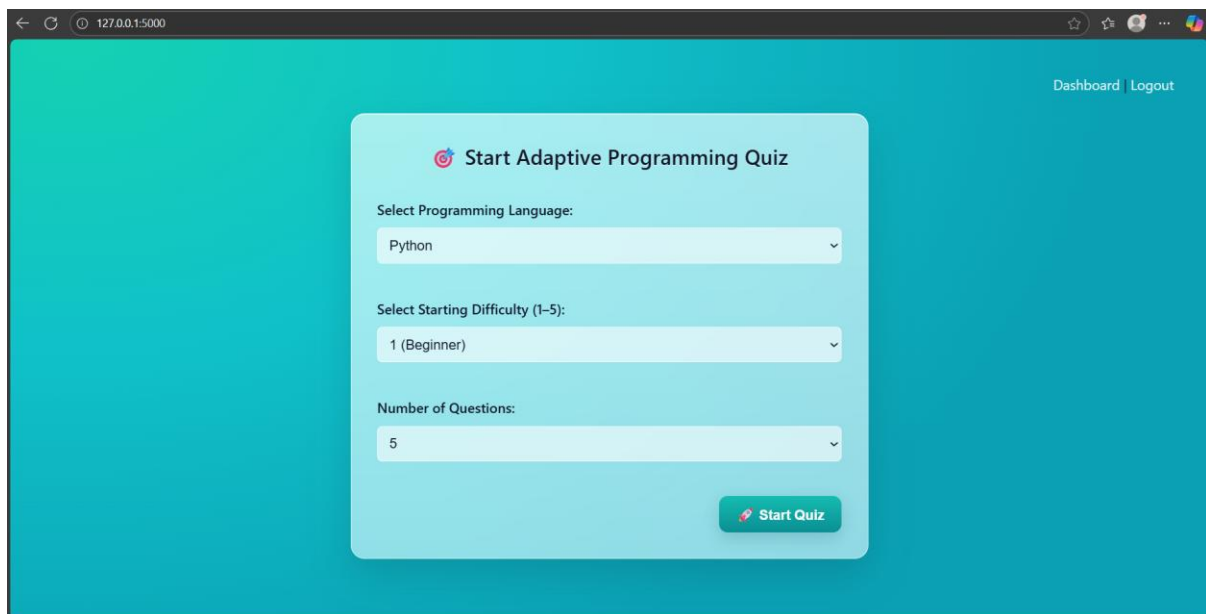
The frontend is deliberately implemented using **server-rendered HTML5 and CSS3 only**, with no client-side JavaScript. Every interaction follows the classic request–response cycle: the learner submits a form or clicks a button, and the server responds with a fully re-rendered page reflecting the new state. This design keeps the experience fast, predictable, and secure, as no API keys or sensitive logic ever leave the server, and there are no concerns about Cross-Origin Resource Sharing (CORS), token handling, or partial UI hydration.

The interface is organised into a small set of pages, each with a clear purpose. **Authentication pages** include `login.html`, which provides a username/password form with inline error banners and links for password reset or guest access, and `signup.html`, which collects credentials and a “remember word” for knowledge-based password recovery. The reset flow is handled by `forgot_password.html` and `forgot_password_step2.html`, which implement a two-step KBA process: users first provide their username, then answer character-position challenges from the remember word before being allowed to reset their password. All validation feedback is rendered server-side directly beneath the relevant inputs.



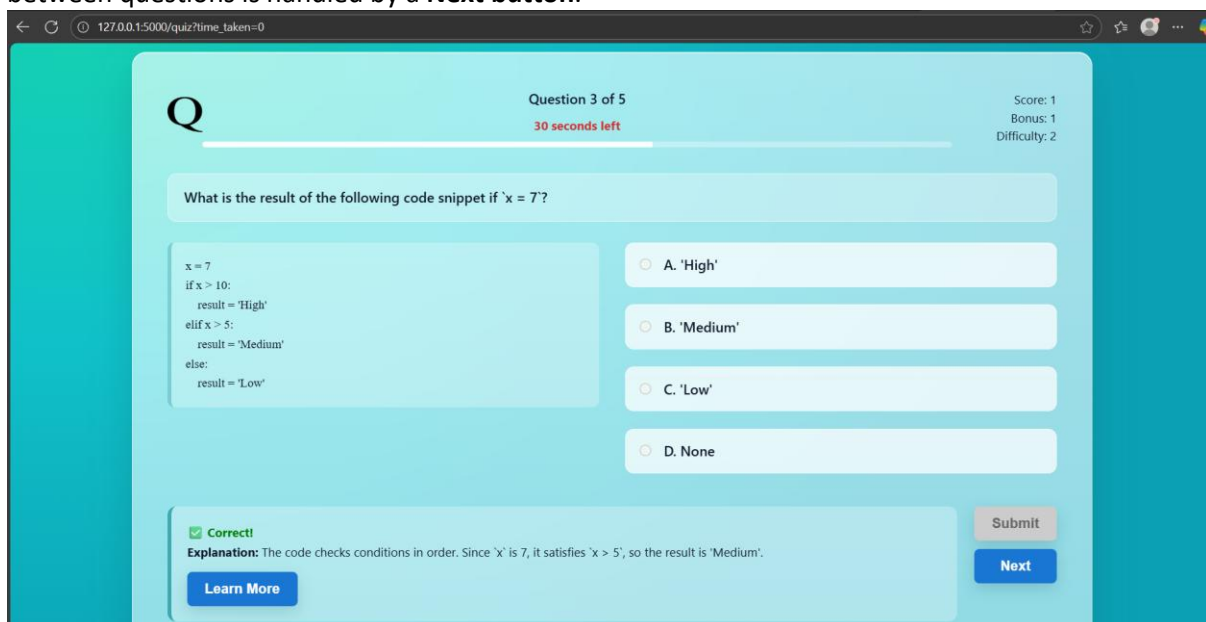
(Figure 12, Login page)

Once authenticated or continued as guest, learners arrive at **home.html**, where they configure a session by selecting a language (Python, Java, or C), starting difficulty, and number of questions., the user is taken into **quiz.html**, the core of the experience.



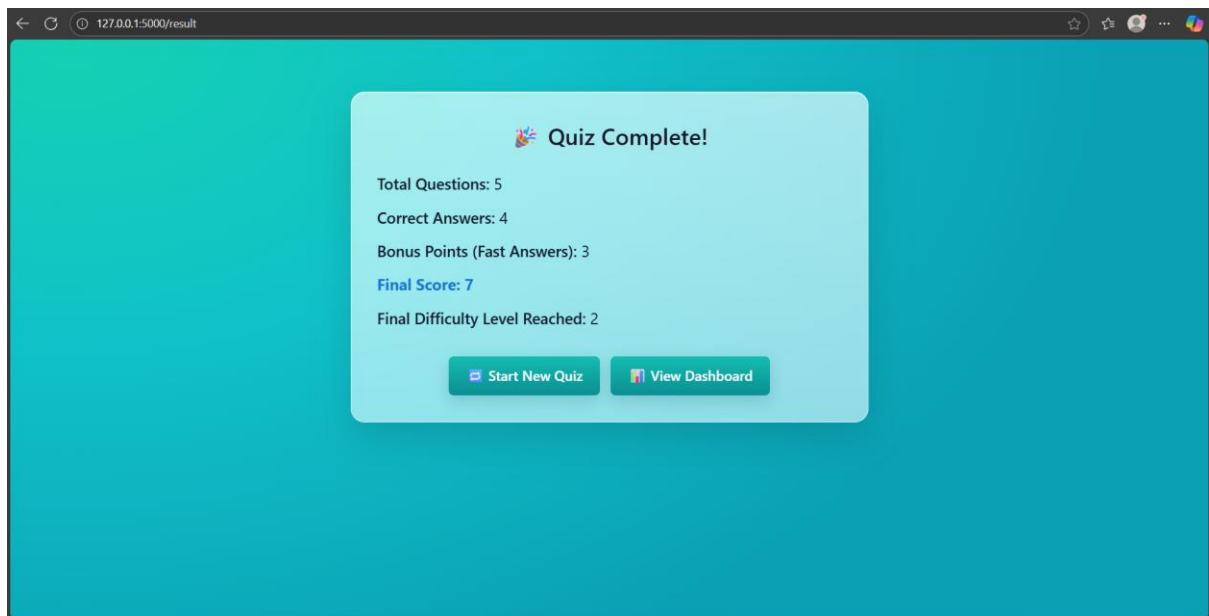
(Figure 13, Home page)

In the Quiz page Questions are displayed with the stem at the top and any code snippet shown in a monospaced block, styled to wrap gracefully for readability. Options are rendered as radio buttons with associated `<label>` tags, maximising both accessibility and usability. Submitting an answer via POST re-renders the same page with a **feedback panel**, a coloured box indicating correctness and a short explanation. Learners may then request further detail using the “Learn more” button, which posts back to the same page and appends an extended explanation below the feedback. Navigation between questions is handled by a **Next button**.



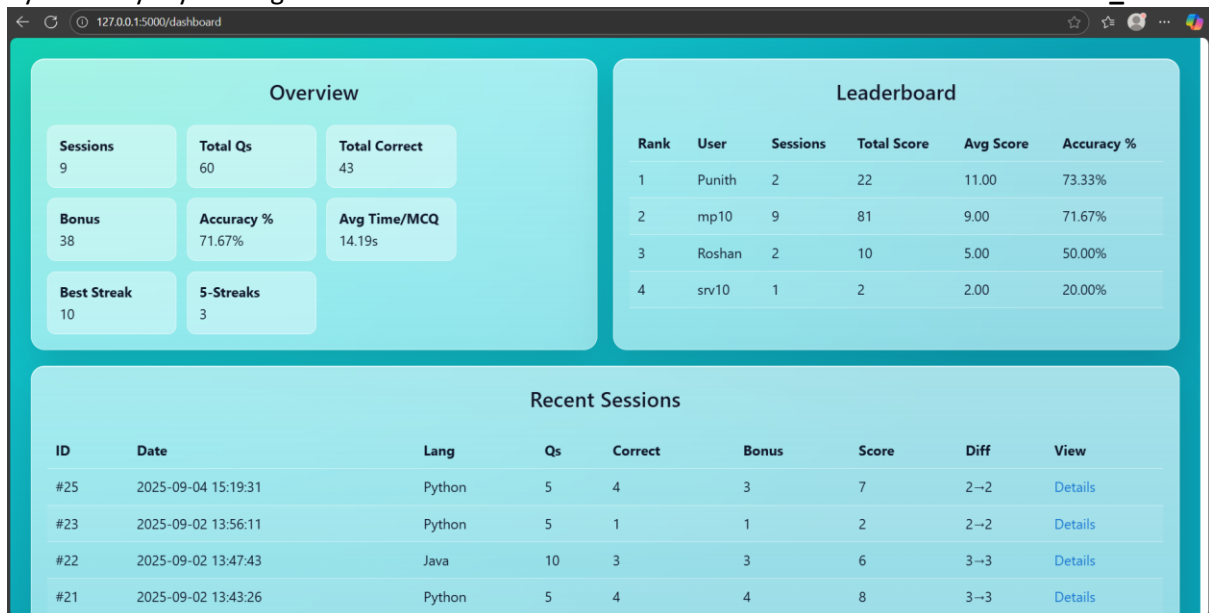
(Figure 14, Quiz page)

Session closure is handled by **result.html**, which provides a concise summary of finished session: total score, number of correct answers, bonus points, final difficulty.



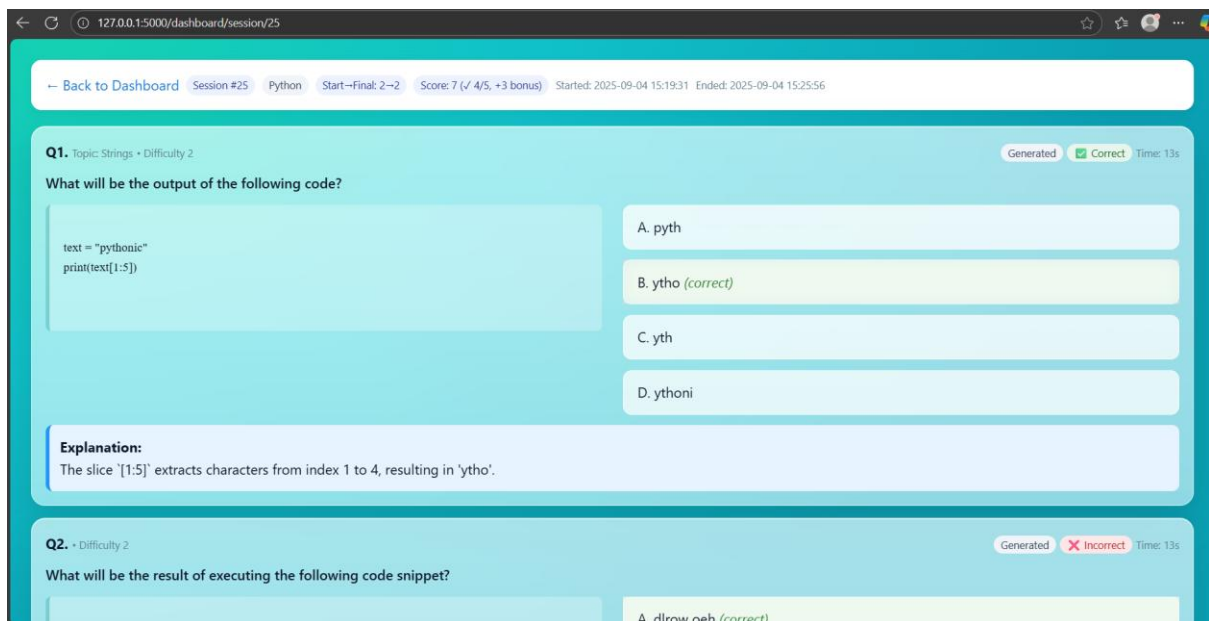
(Figure 15, Results page)

Longer-term tracking is available in **dashboard.html**, which aggregates recent sessions, showing accuracy by topic and difficulty. It also has a global leaderboard which displays top five Users ranked by accuracy. By clicking on “Details” in the Recent Sessions Card user is redirected to **session_detail**.



(Figure 16, User Dashboard page)

A drill-down view, **session_detail.html**, lists each question from a session with its topic, difficulty, user's answer, correct option, correctness, and time taken, formatted consistently with the quiz interface.



(Figure 17, Recent Sessions page)

The interface uses a calm **teal-cyan gradient** background with **frosted-glass cards**. This “glassmorphism” gives clear visual hierarchy—forms sit on light, airy containers—while keeping focus on the inputs. **Uniformity** is maintained across pages through a single spacing scale, consistent button styles (primary/secondary), identical form layouts. **Readability** is prioritised with generous line height, a clean system font stack for text, and a monospaced font for code blocks; labels are paired with every input, and large click targets improve accessibility. Overall, the UI is modern, lightweight, and consistent—quick to scan, easy to use, and visually cohesive from login to results.

7.8 Flask Endpoints & Flow

7.8.1 Authentication and Profile Management

The system provides standard routes for user management.

- GET /signup renders the registration form, while POST /signup creates a new account by hashing the password and storing a “remember word” for knowledge-based resets.
- GET /login renders the login form; POST /login verifies credentials and, on success, initialises the Flask session with loggedin=True and the user_id, redirecting the learner to /home.
- GET /logout clears the session and redirects to the login page.

Password reset is handled in two steps. GET /forgot-password asks for a username; POST /forgot-password checks existence and then redirects to the next stage. GET /forgot-password-step2 renders a challenge asking for specific characters of the remember word. POST /forgot-password-step2 verifies the input, updates the stored (hashed) password, and redirects to /login. All error states are re-rendered inline with explanatory messages.[1]

```

@app.route("/signup", methods=["GET", "POST"])
def signup():
    msg = ''
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        remember_word = request.form["remember_word"]

        cursor.execute("SELECT * FROM users WHERE username=%s", (username,))
        account = cursor.fetchone()
        if account:
            msg = "Username already exists!"
        else:
            hashed_password = generate_password_hash(password)
            cursor.execute(
                "INSERT INTO users (username, password, remember_word) VALUES (%s, %s, %s)",
                (username, hashed_password, remember_word)
            )
            db.commit()
            return redirect(url_for("login"))
    return render_template("signup.html", msg=msg)

@app.route("/login", methods=["GET", "POST"])
def login():
    msg = ''
    if request.method == "POST":
        username = request.form["username"]
        password = request.form["password"]
        cursor.execute("SELECT * FROM users WHERE username=%s", (username,))
        account = cursor.fetchone()
        if account and check_password_hash(account["password"], password):
            session["loggedin"] = True
            session["username"] = account["username"]
            return redirect(url_for("index"))
        msg = "Incorrect username or password!"
    return render_template("login.html", msg=msg)

```

7.8.2 Session Setup

After authentication, learners configure their quiz session. GET /home presents a form for choosing the programming language, starting difficulty, and number of questions. When submitted, POST /start creates a new row in quiz_sessions, records metadata such as language and start difficulty, resets counters in the adaptive engine, and clears the set of seen example IDs. Control then passes directly to the quiz loop via a redirect to /quiz.

```

@app.route("/quiz", methods=["GET", "POST"])
def quiz():
    if not ("loggedin" in session or "guest" in session):
        return redirect(url_for("login"))

    session_id = session.get("session_id")
    engine = active_sessions.get(session_id)

    if not engine or not engine.has_more_questions():
        return redirect("/result")

```

7.8.3 Quiz Loop

After setting up the Quiz page the quiz loop is expressed through three routes:

- **Display (GET /quiz)**

This route never generates questions; it only consumes what the engine has already staged. If a difficulty proposal is pending, the server renders quiz.html with an overlay modal for *Accept* or *Keep*. Otherwise, if a staged item exists, it is consumed and displayed. If prefetch is still running, the server blocks on the engine's Event until the item is ready. Ordering is guaranteed, and the learner never sees duplicates or half-formed states.

- **Submission (POST /quiz)**

When the learner submits an answer, the system records a row in quiz_attempts, updates aggregates in quiz_sessions (score, streaks, bonus points), and advances the adaptive engine's state. If the stage was "same," it transitions to "alt"; if "alt," it resets to "same" with a new example. Immediately after submission, the engine starts a background prefetch of the next question (generation → validation → fallback). The page re-renders with a feedback panel showing correctness and a brief explanation, along with buttons for *Know more* and *Next*.

- **Difficulty decisions (POST /quiz/decide)**

When the modal is displayed, the learner's choice is submitted here. If accepted, the difficulty shifts by ± 1 (clamped to 1–5); if kept, the proposal is discarded. In both cases, the pending flag is cleared, counters adjusted, and the system resumes normal prefetching. The user is then redirected to GET /quiz, which consumes the next staged item.

Additional enrichment is provided by POST /learn_more, which calls the learner module with the just-answered item and outcome, and re-renders quiz.html with an extended explanation appended beneath the feedback panel. This does not advance the stage or alter prefetching.

```
@app.route("/learn_more", methods=["POST"])
def learn_more():
    if not ("loggedin" in session or "guest" in session):
        return {"ok": False, "error": "auth"}, 401

    session_id = session.get("session_id")
    engine = active_sessions.get(session_id)
    if not engine or not engine.current_mcq:
        return {"ok": False, "error": "no-mcq"}, 400

    # Prefer payload answer; fall back to saved last answer
    data = request.get_json(silent=True) or {}
    user_answer = (data.get("answer") or session.get("last_answer") or "").strip()

    try:
        result = lm_generate(engine.current_mcq, user_answer)
        if not result:
            return {"ok": False, "error": "generation-failed"}, 500
        return {"ok": True, "text": result.get("text", "").strip()}
    except Exception as e:
        print("Learn More route error:", e)
        return {"ok": False, "error": "exception"}, 500
```

7.8.4 Finishing and Analytics

When the target number of questions has been answered, GET /result finalises the session by updating quiz_sessions with ended_at and the final_difficulty. The rendered page provides a session summary including score, bonus, streaks, and a table of recent items for quick review. Longer-term tracking is handled by GET /dashboard, which aggregates recent sessions and shows accuracy by topic and difficulty as simple CSS progress bars or tables. GET /dashboard/session/<id> drills down into a single session, listing each item with difficulty, answer, correctness, and timing.

7.8.5 Error Handling and Fallbacks

The system's error-handling contract is simple: the learner never sees a broken state. If OpenAI generation or validation fails during prefetch, the system stages the vetted example E as a fallback and signals the Event, so GET /quiz still renders without delay. If a difficulty proposal is pending, prefetch halts and the overlay is shown immediately on the next GET. All form validation errors re-render the same page with inline messages, while structural fallbacks (e.g., malformed MCQ JSON) are invisible to the learner.

7.9 Persistence Layer

The persistence layer is implemented in **MySQL (InnoDB, utf8mb4)** and acts as the authoritative store for all user accounts, quiz sessions, and per-question attempts. **Users** table stores the user log in details, quiz_sessions and quiz_attempts store quiz performance with user id as foreign key. This database design ensures that the tutor can reliably resume progress, compute results accurately, and generate analytics for dashboards. The schema is deliberately small and normalised—users → quiz_sessions → quiz_attempts—so that writes remain simple and reads remain predictable.

7.9.1 Tables Creation:

```
-- users
CREATE TABLE users (
  id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(64) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL,      -- hashed
  remember_word VARCHAR(64) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- quiz_sessions
CREATE TABLE quiz_sessions (
  id INT AUTO_INCREMENT PRIMARY KEY,
  user_id INT NOT NULL,
  language ENUM('python','java','c') NOT NULL,
  difficulty TINYINT NOT NULL,          -- starting difficulty (1..5)
  num_questions INT NOT NULL,
  score INT DEFAULT 0,
  correct_answers INT DEFAULT 0,
  bonus_points INT DEFAULT 0,
  streak INT DEFAULT 0,
  five_streaks INT DEFAULT 0,
  final_difficulty TINYINT DEFAULT NULL,
  started_at DATETIME NOT NULL,
  ended_at DATETIME DEFAULT NULL,
  CONSTRAINT fk_sessions_user
    FOREIGN KEY (user_id) REFERENCES users(id)
    ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
-- quiz_attempts
CREATE TABLE quiz_attempts (
  id INT AUTO_INCREMENT PRIMARY KEY,
  session_id INT NOT NULL,
  question_id VARCHAR(64) NULL,          -- dataset id; NULL for AI
  is_ai_generated BOOLEAN NOT NULL DEFAULT 0,
  user_answer CHAR(1) NOT NULL,          -- 'A'..'D'
  correct_option CHAR(1) NOT NULL,        -- 'A'..'D'
  is_correct BOOLEAN NOT NULL,
  topic VARCHAR(64) NOT NULL,
  difficulty TINYINT NOT NULL,
  time_taken DECIMAL(6,2) NOT NULL,      -- seconds
  CONSTRAINT fk_attempts_session
    FOREIGN KEY (session_id) REFERENCES quiz_sessions(id)
    ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

7.9.2 Write Paths

Database interactions occur at well-defined points in the tutor workflow:

- **/signup** inserts a new row into users with a hashed password and remember word.

```
cursor.execute("SELECT * FROM users WHERE username=%s", (username,))
account = cursor.fetchone()
if account:
    msg = "Username already exists!"
else:
    hashed_password = generate_password_hash(password)
    cursor.execute(
        "INSERT INTO users (username, password, remember_word) VALUES (%s, %s, %s)",
        (username, hashed_password, remember_word)
    )
db.commit()
```

- **/start** creates a new entry in quiz_sessions with the selected language, starting difficulty, question count, and timestamp.

```
cursor.execute(
    """INSERT INTO quiz_sessions
    (user_id, username, language, start_difficulty, final_difficulty,
    total_questions, correct, bonus, final_score, started_at,
    longest_streak, five_streaks)
    VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)""",
    (
        user_id,
        username,
        language,
        int(difficulty),
        int(difficulty),
        int(total_questions),
        0, # correct
        0, # bonus
        0, # final_score
        datetime.utcnow(),
        0, # longest_streak
        0 # five_streaks
    )
)
db.commit()
```

- **/quiz (POST)** records each answered question by inserting into quiz_attempts and updating aggregates in quiz_sessions (score, correct answers, bonus points, streaks).

```

# ----- Persist this attempt -----
db_session_id = session.get("db_session_id")
if db_session_id:
    source_type = "example" if mcq.get("id") else "generated"
    topic = mcq.get("topic") or None
    q_index = engine.count # incremented in submit_answer()

    cursor.execute(
        """INSERT INTO quiz_attempts
        (session_id, q_index, source_type, topic, difficulty, is_correct,
        time_taken_sec, correct_option, user_answer, mcq_json)
        VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)""",
        (
            db_session_id,
            q_index,
            source_type,
            topic,
            engine.current_difficulty,
            bool(correct),
            int(time_taken),
            mcq["correct_option"],
            request.form["answer"],
            json.dumps(mcq, ensure_ascii=False)
        )
    )
    db.commit()

```

- **/quiz/decide** updates session difficulty and counters if a user accepts a difficulty change.
- **/result** finalises the session by updating ended_at and final_difficulty.

All queries are executed with parameterised statements through **flask_mysqldb**, preventing injection and keeping transactions lightweight. Since the adaptive engine operates in memory, there are no database reads in the hot path of prefetch; persistence only occurs on submission.

And to keep this as production ready version currently this system's MySQL instance runs on **Railway** [5], a managed PaaS. This choice keeps the app's server lightweight while providing a durable, internet-accessible DB with simple environment-variable configuration. So this makes the application a runnable ready code without any need of local database config set up or tables creation.

Configuration used by the app are set in .env; they're read by app.py. If anyone prefer to run with own database, they can easily parse that at startup and populate the variables below (in .env).

```

MYSQL_HOST=<railway-host>
MYSQL_PORT=<railway-port>
MYSQL_USER=<railway-user>
MYSQL_PASSWORD=<railway-password>
MYSQL_DB=<railway-database>

```

7.10 Dashboard and Analytics

The dashboard provides learners with a consolidated view of their performance across multiple quiz sessions. Its purpose is both diagnostic—helping learners understand their strengths and weaknesses—and motivational, by presenting progress in a clear, interactive format. The analytics layer is powered by aggregated data from quiz_sessions and quiz_attempts, rendered through Flask templates and client-side charting libraries.

7.10.1 Overview metrics

At the top of the dashboard, high-level statistics summarise cumulative performance. These include the total number of sessions completed, total questions attempted, the number answered correctly, bonus points earned, and overall accuracy expressed as a percentage. Time-based metrics such as the

average time taken per question add a measure of efficiency, while streak counters highlight longest and most frequent runs of consecutive correct answers. These headline figures give learners a quick sense of progress and consistency.

7.10.2 Leaderboards

To provide social and motivational context, a leaderboard component ranks users across the system. Sorting is based on accuracy or final scores aggregated from all sessions, with the top performers displayed alongside their usernames. This feature is particularly effective in a classroom or cohort setting, where visibility of peer performance can encourage sustained engagement and competition.

7.10.3 Session history

A “recent sessions” panel lists the most recent quiz attempts (typically the last five), including language, difficulty progression, number of questions, and final score. This allows learners to revisit their most recent activity and see whether performance is improving over time. Each row links to a detailed session view, where individual MCQs—including the learner’s answers, correct options, and explanations—are replayed for review.

7.10.4 Topic and difficulty analytics

Two interactive graphs provide deeper insight into strengths and weaknesses.

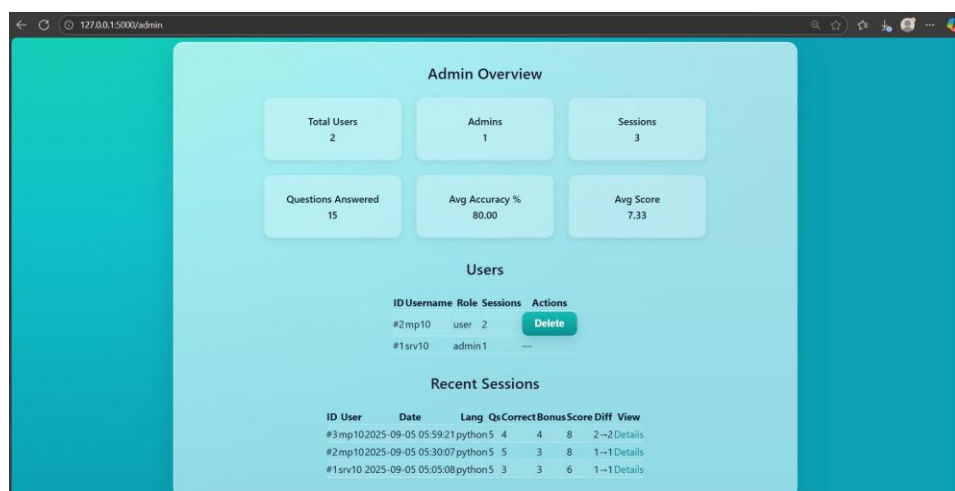
Topic accuracy: a bar chart shows percentage correctness grouped by topic. A dropdown filter allows learners to view topic-level performance within a specific programming language (Python, Java, or C).

Difficulty accuracy: a line chart plots accuracy against difficulty levels (1–5), also filtered by language. This reveals whether learners handle basic constructs comfortably but struggle with higher-order concepts, or vice versa.

Together, these views help identify specific areas for targeted revision.

7.10.5 Admin Dashboard

This is proposed in the final stage of development hence it was partially developed just to explore the possibilities of a admin access which can be scaled for teacher and tutors in future scope. This is global dashboard and the admin can view all Users performances and recent sessions. Admin can also delete an User profile.



Since it is not yet completely developed and tested, it is not mentioned in the features list.

7.11 Security

7.11.1 Secrets and Environment Management

All sensitive configuration is externalised into **environment variables**, with a `.env` [9] file used in development and real environment variables set in production. Key variables include:

- `FLASK_SECRET_KEY` — used exclusively for signing Flask session cookies and protecting against tampering.
- `OPENAI_API_KEY` — consumed server-side by the generator, validator, and learner modules; never exposed to the client.
- `MYSQL_HOST`, `MYSQL_PORT`, `MYSQL_USER`, `MYSQL_PASSWORD`, `MYSQL_DB` — credentials for the hosted MySQL instance (Railway).

Best practices are enforced throughout: the `.env` file is git-ignored to prevent accidental leaks, and no secrets are ever embedded in templates or transmitted to the browser. In the event of a compromise, keys can be rotated immediately without code changes, ensuring resilience and minimising risk.

7.11.2 Authentication and Password Safety

User credentials are handled according to modern security standards. Passwords are never stored in plaintext; instead, they are hashed and verified using the **werkzeug.security** utilities, which implement safe, salted algorithms.

Password resets avoid email-based flows (to reduce dependency on external systems) and instead use a knowledge-based mechanism. Each user sets a “remember word” during signup, which is later checked via positional challenges during the reset flow. When a password is reset, the new value is hashed again before being persisted.

8. Challenges

8.1 Creating & Labelling Datasets

The curated examples (E) are the **anchor** of the entire tutor. They seed generational items, provide ground-truth topics and difficulty labels, and act as the **fallback** whenever generation or validation fails. Any mislabelling (e.g., a “3” that behaves like a “5”) would **break adaptivity**, confuse the validator, and degrade explanations.

So, the dataset creation were guided by publicly available learning resources and tutorials, such as Python.org, GeeksForGeeks, W3Schools, and Real Python and Custom Knowledge via LLM. And all the code snippets are again manually tested to validate the output against answers. These helped ensure coverage consistency, pedagogical relevance, and technical accuracy.[24][25][26]

8.2 GPT Hallucinations

An assessment/Educational platform system cannot tolerate inaccuracies or inconsistencies in learning. These hallucinations create ambiguous answers which erode trust and learning value. So, we needed a way to use LLM creativity **without** sacrificing correctness.

We constrained generation with **schema-first prompts** (stem, optional code, four options A–D, one correct_option, short explanation) and set conservative parameters. Every candidate then passes an **independent validator** that re-derives the correct answer and returns a **binary** verdict. For the validator the “temperature” parameter is set to “0” so it can be minimum creative and avoid hallucinations during validation. While in the other hand generator is given freedom to be creative.

8.3 User Experience & Latency Between Questions

Learners need a smooth rhythm: answer → feedback → next. LLM calls introduce variable latency, and early tests showed a time gap of 4-5 seconds between the transmission. This disengages the learner and keep user await.

Since the GPT calls and generation time cannot be minimised much, I introduced **background prefetch** immediately after submission (while the learner reads feedback) and a background thread computes the next item, **stages** it, and keep it ready to display as soon as the User clicks Next button. So, without heavy weight client side computing the latency is minimised my using relatively easy Multi-threading concept to make the UX snappy.

8.4 Testing a Non-Deterministic, Data-Driven System

This tutor isn't a classic system with fixed "expected outputs." Each run can differ because examples are selected randomly and LLM generations vary. That non-determinism makes fully automated testing hard: unit tests can check structure, but they can't guarantee **pedagogical correctness** or catch subtle ambiguities that undermine learning. Without a solid testing approach, reliability and trust would be at risk.

To overcome this, combined automation with deliberate **manual evaluation** (mostly manual testing). For internal testing, I used **fixed seeds** for example selection and controlled prompt inputs, so I could re-run the same sessions when investigating issues. Because quality is ultimately pedagogical, I **manually inspected every generated MCQ** used in test runs. I checked for single-objective stems, plausible distractors (no co-correct options), language accuracy, and explanation clarity.

Even with non-determinism, this layered strategy—produced a tutor that serves only valid questions, adapts as intended, and feels fast in use.

9. Testing & Evaluation

9.1 Objectives and Approach

Because both item selection and language model outputs are non-deterministic, the testing strategy adopted a **hybrid approach**. Automated checks were used to enforce schema validity, option normalisation (A–D), and difficulty bounds, while the validator acted as a gate to ensure that only unambiguous items reached the learner. For reproducibility during debugging, controlled prompts and fixed seeds were employed. In addition, manual review was performed on generated items to evaluate pedagogical correctness—ensuring that each MCQ targeted a single objective, contained one defensible answer, included plausible distractors, and maintained programming-language accuracy. The evaluation covered **all three languages (Python, Java, C)** across the full range of difficulties (1–5).

9.2 Core Quiz Engine (Fetcher, Generator, Validator)

The quiz engine was tested at three layers:

- **Fetcher accuracy.** The fetcher is a purely logical unit. It consistently returned correct, unseen examples at the requested difficulty, with edge cases also validated. Accuracy was therefore measured at **100%**.
- **Generator performance.** The generator's output was assessed independently to capture system efficiency. Out of 60 generated MCQs reviewed across languages and difficulties, **54 were valid (90%)**.

- **System accuracy (Generator + Validator + Fallback).** When combined with the validator, which blocks invalid items while still allowing valid ones to pass, the effective accuracy rose to **95%**. This demonstrates the intended guardrail effect: ambiguous or malformed generations were rejected, and the pipeline fell back to curated examples, so the learner never encountered a broken item.

This confirms that generator performance defines efficiency, but generator plus validator defines **true system reliability**.

9.3 Endpoint and Integration Testing

As the frontend was integrated, each Flask endpoint was exercised with realistic data to validate the full workflow:

- **Authentication and reset:** /signup, /login, /logout, and the two-step /forgot-password reset (with remember-word position checks).
- **Session flow:** /start opened a new quiz session; GET /quiz displayed staged questions or waited deterministically; POST /quiz persisted attempts and triggered background prefetch; /quiz/decide applied or rejected difficulty proposals; /result finalised the session.
- **Explanations:** /learn_more returned extended feedback without altering engine state.
- **Dashboards:** /dashboard and /dashboard/session/<id> read directly from MySQL to display recent sessions, topic accuracy, and difficulty-based performance.

9.4 Takeaways

Testing showed that the combination of curated datasets and validator-gated generation yielded **95% valid items served**, demonstrating that the guardrails were effective in practice. The **prefetch and single-flight design** eliminated visible delays and ensured strict ordering, significantly improving responsiveness without relying on client-side JavaScript. Endpoint behaviour was predictable under realistic use, confirming that the web layer cleanly encapsulated the adaptive engine's state machine.

Overall, the evaluation indicates that the system is both **robust and pedagogically reliable**, balancing novelty from AI generation with safeguards that guarantee quality.

10. Conclusion

This project delivers a functional **AI-powered programming MCQ tutor** that is reliable, adaptive, and explainable. The system combines a curated dataset with a controlled LLM-driven pipeline, safeguarded by a validator-and-fallback mechanism, and an adaptive engine that modulates difficulty in a transparent two-step loop (*same* → *conditional up/down*). The architecture is intentionally minimal—Flask for the backend, server-rendered HTML/CSS for the frontend, and MySQL (hosted on Railway) for persistence—so that most engineering effort is focused on maximising learning value through item quality, adaptivity, and feedback.

Key Achievements

- **Quality under control.** The validator independently re-derives answers and enforces the “exactly one correct option” rule. When a generation is ambiguous, the system falls back to curated examples, ensuring learners never encounter a broken item. Testing across three languages and all difficulty levels showed generator-only accuracy at 90%, while end-to-end system accuracy reached 95%—evidence of the validator’s effectiveness in both blocking invalid items and allowing valid ones.
- **Adaptive without surprises.** The state machine preserves topical coherence by deriving conditional questions (C) from the same example (E) which is beneficial in either of the situations. If the user answers right (c) helps to deepen his knowledge in the same concept else (c) helps to understand the basic underpinning of the concept. And also difficulty adjustments are proposed only at stable points and require explicit user confirmation, with difficulty always clamped between 1 and 5.
- **Smooth user experience.** Background prefetching ensures that the *Next* button is display-only (wait & consume), eliminating duplicate or out-of-order questions and reducing perceived latency. This is achieved without any client-side JavaScript, keeping the frontend lightweight and robust.
- **Sound data model and persistence.** A normalised relational schema (users → sessions → attempts) supports dashboards, summaries, and drill-downs, while storing minimal PII and maintaining strong integrity constraints.
- **Testing in a non-deterministic setting.** A hybrid evaluation strategy—combining automated structural checks, validator gating, reproducible seeded runs, and manual review—proved necessary and sufficient to validate correctness and adaptivity under conditions of randomness in both dataset sampling and LLM output.

Limitations and Outlook

The current system’s topical breadth is limited by dataset size, LLM outputs retain some variability, and difficulty adjustments are heuristic rather than psychometrically calibrated. Despite these constraints, the system achieves its central goal: a dependable, low-latency tutor that adapts to learner performance, safeguards quality, and provides succinct explanations.

This foundation is strong enough to support **future work**, such as expanding datasets for broader coverage, exploring richer explanation strategies, refining adaptivity with more sophisticated models, and conducting formal user studies to measure learning gains.

11. Future Work

The current system demonstrates that adaptive, validator-gated AI generation can power a reliable programming tutor. Several extensions would strengthen its impact, broaden its coverage, and improve usability. The following directions represent practical next steps building on the existing codebase.

11.1 Content and Coverage

A first priority is to expand the curated datasets. The current base of ~100 items per language is sufficient for prototyping but limits topical breadth. Scaling up would involve adding new programming languages (e.g., C++, JavaScript) and extending into higher-level domains such as data structures, algorithms, and object-oriented design patterns. Authoring tools could also be introduced to make dataset creation and labelling more efficient.

Another enhancement is to build **item banks by fine-grained objective**. Rather than grouping only by language and difficulty, items could be categorised by specific skills—such as Python slicing with strides or Java method overriding edge cases. This would improve both adaptive targeting and analytics, enabling learners to track mastery of individual concepts.

11.2 User Experience and Accessibility

The user interface could be refined for a **mobile-first experience**, optimising touch targets, font scaling, and session continuity in offline-friendly scenarios. A full accessibility pass would ensure inclusivity: adding ARIA roles, conducting contrast audits, benchmarking keyboard-only navigation, and optionally providing text-to-speech for question stems.

Dashboards could also be extended with richer visualisations. While the current CSS-only progress bars are intentionally lightweight, minimal client-side JavaScript libraries (e.g., Chart.js) could be selectively introduced to provide interactive charts where stakeholders value deeper insights.

11.3 Administrative Tools

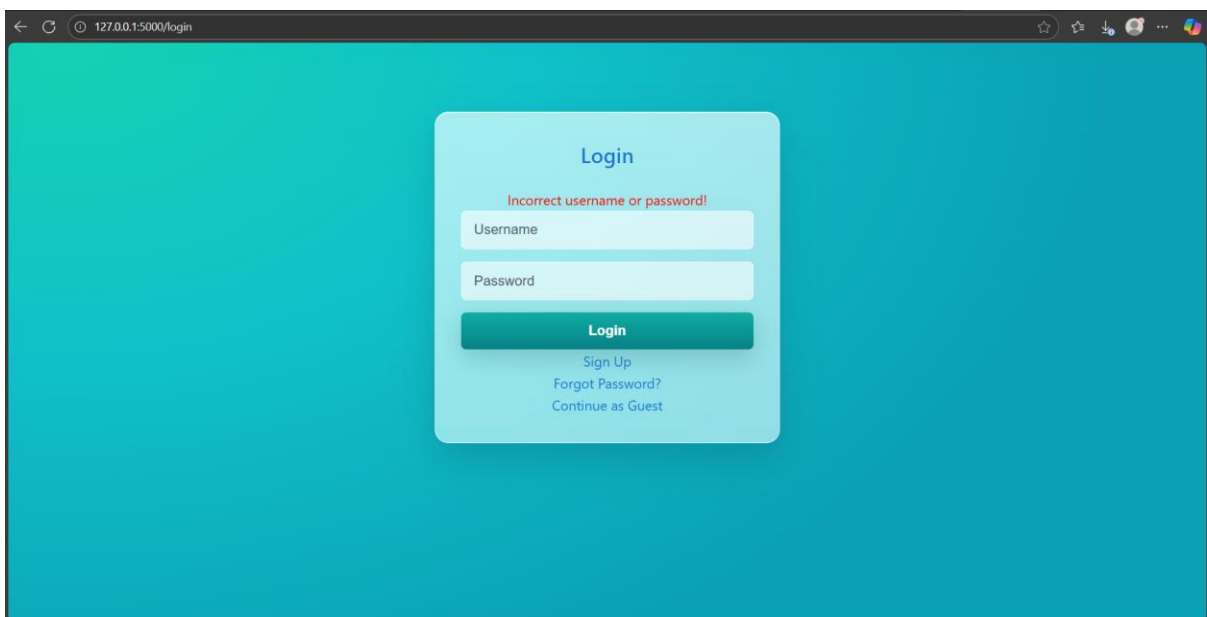
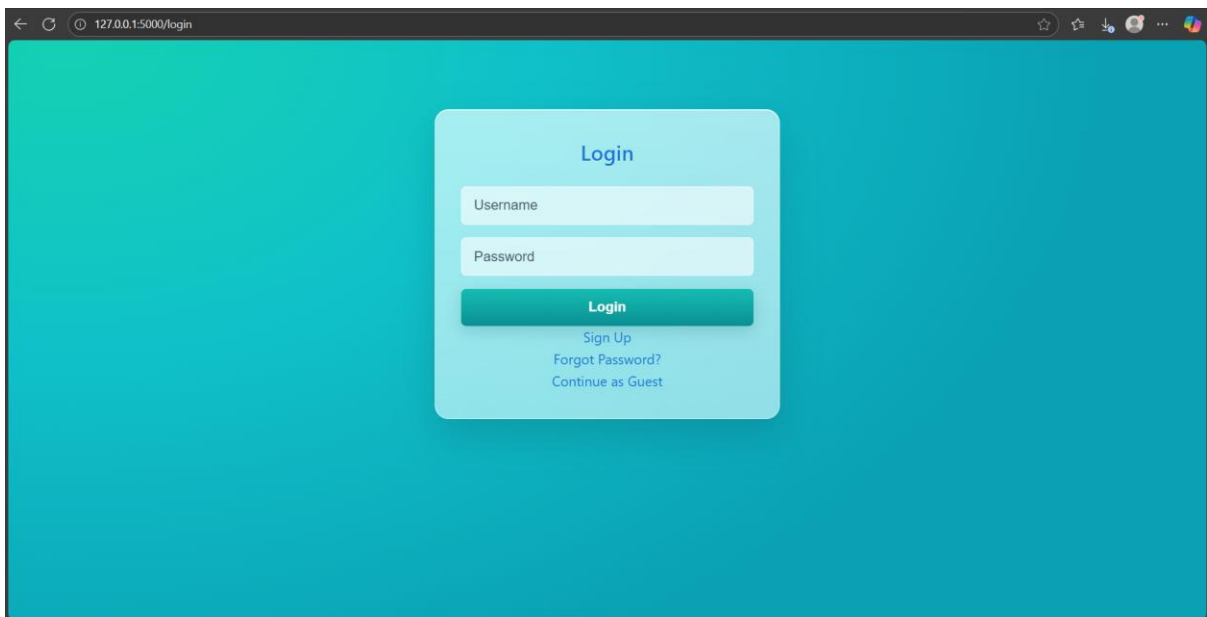
A final avenue is the introduction of **admin access** for content management. This idea is partially implemented. This would allow authorised users to add new datasets, introduce new programming languages, and adjust difficulty mappings directly from the web interface. Such tools would make the system easier to maintain and scale, supporting long-term use in educational settings.

References:

1. Pallets Projects, Flask Documentation — <https://flask.palletsprojects.com/>
2. Pallets Projects, Jinja2 Documentation — <https://jinja.palletsprojects.com/>
3. Pallets Projects, Werkzeug Security — Password Hashing Utilities — <https://werkzeug.palletsprojects.com/en/stable/utils/#security-helpers>
4. Oracle, MySQL Reference Manual — <https://dev.mysql.com/doc/>
5. Railway, Railway MySQL Service — Product Docs — <https://docs.railway.com/guides/mysql>
6. OpenAI, API Reference – Chat Completions — <https://platform.openai.com/docs/api-reference/chat>
7. OpenAI, API Reference – Responses — <https://platform.openai.com/docs/api-reference/responses>
8. OpenAI, Python SDK — libraries page — <https://platform.openai.com/docs/libraries>
9. python-dotenv — User Guide — <https://saurabh-kumar.com/python-dotenv/>
10. Flask-MySQLdb — Extension Documentation — PyPI: <https://pypi.org/project/Flask-MySQLdb/>
11. GitHub: <https://github.com/alexferl/flask-mysqldb>
12. mysqlclient (MySQLdb) — Driver Documentation — <https://mysqlclient.readthedocs.io/>
13. Chart.js — Documentation — <https://www.chartjs.org/docs/>
14. OpenAI, *GPT-4 Technical Report*, 2023.
15. Zawacki-Richter, O., et al., “Systematic review of research on artificial intelligence applications in higher education,” *International Journal of Educational Technology in Higher Education*, 2019.
16. Kasneci, E., et al., “ChatGPT for Good? On Opportunities and Challenges of Large Language Models for Education,” *Learning and Individual Differences*, 2023.
17. Wainer, H., ed., *Computerized Adaptive Testing: A Primer*, 2nd ed., 2000.
18. Lord, F. M., *Applications of Item Response Theory to Practical Testing Problems*, 1980.
19. Rasch, G., *Probabilistic Models for Some Intelligence and Attainment Tests*, 1960.
20. [ResearchGate](#)[Massachusetts Institute of Technology](#)[SAGE Journals](#)
21. [Routledge](#)
22. [Taylor & Francis](#) [Onlinenbme.org](#)
23. [arXiv](#)[BioMed Central](#)[PubMed Central](#)[ACL Anthology](#)[ACM Digital Library](#)
24. W3Schools – Python https://www.w3schools.com/python/?utm_source=chatgpt.com
25. GeeksForGeeks – <https://www.geeksforgeeks.org/python-mcq/>
26. Stack Overflow Discussions -<https://stackoverflow.com>
27. https://digitalcommons.kennesaw.edu/cday/Spring_2024/Undergraduate_Research/1/?utm_source=chatgpt.com

Appendices

A. UI screenshots



127.0.0.1:5000/signup

Create an Account

Username

Password

Re-enter Password

Remember Word

Re-enter Remember Word

Sign Up

Already have an account? [Log in](#)

127.0.0.1:5000/signup

Create an Account

mp10

Remember Word entries do not match.

Sign Up

Already have an account? [Log in](#)

127.0.0.1:5000/signup

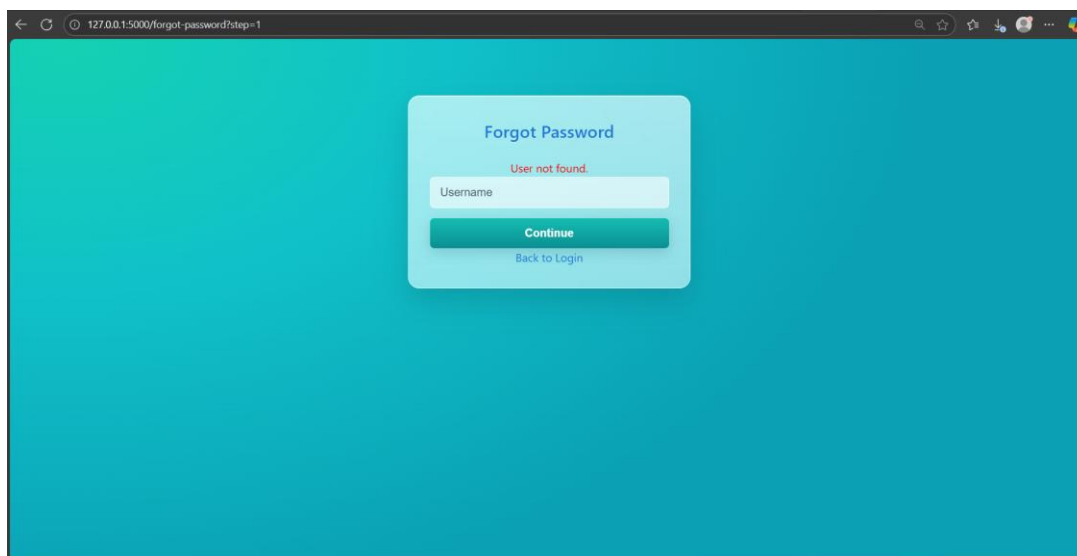
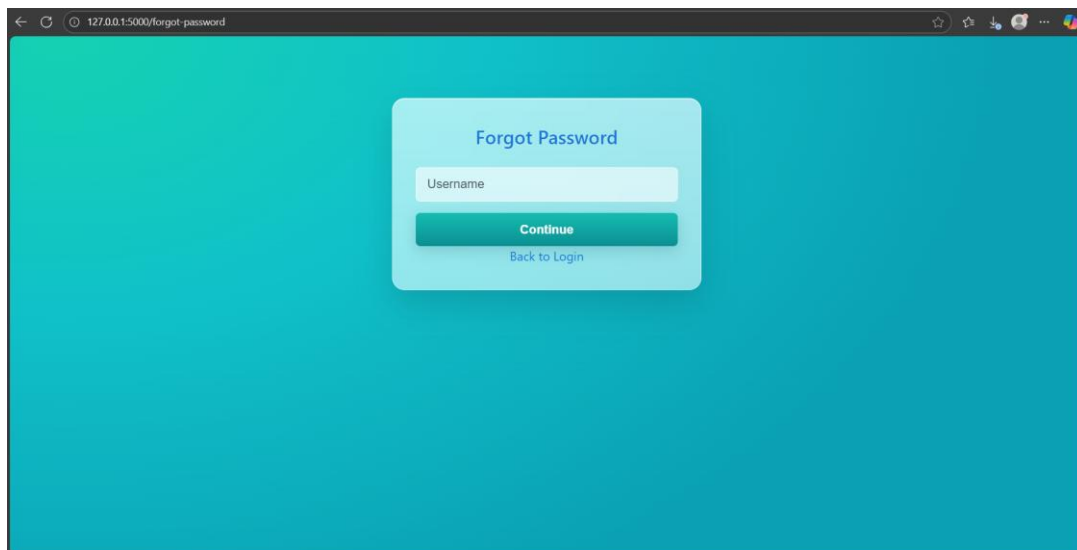
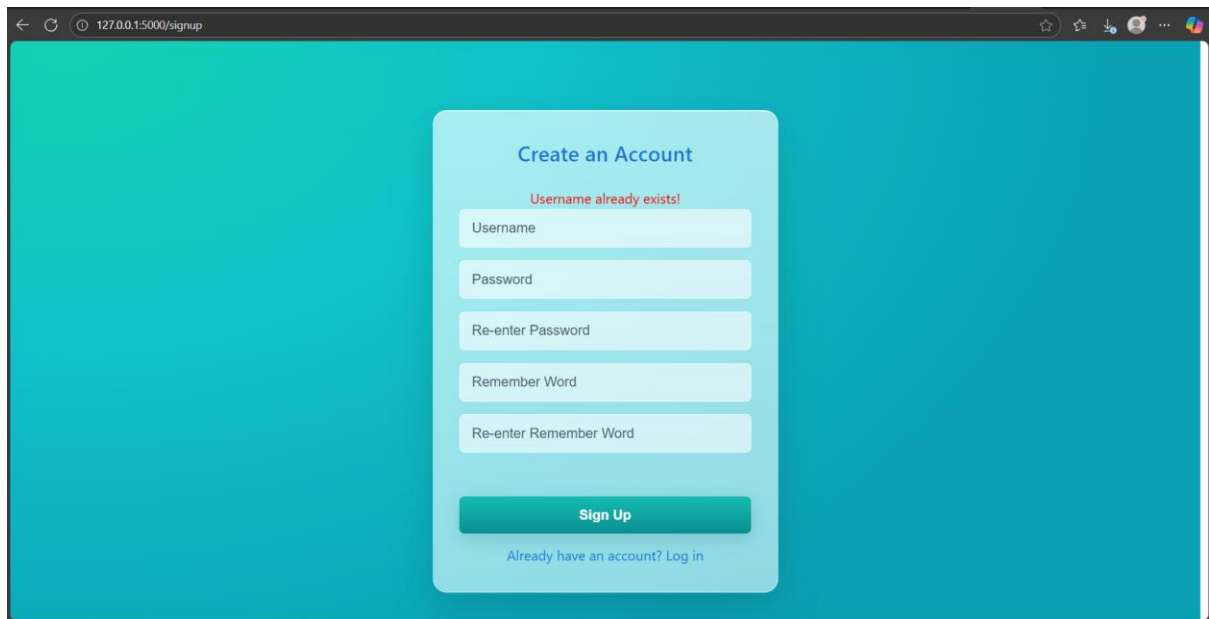
Create an Account

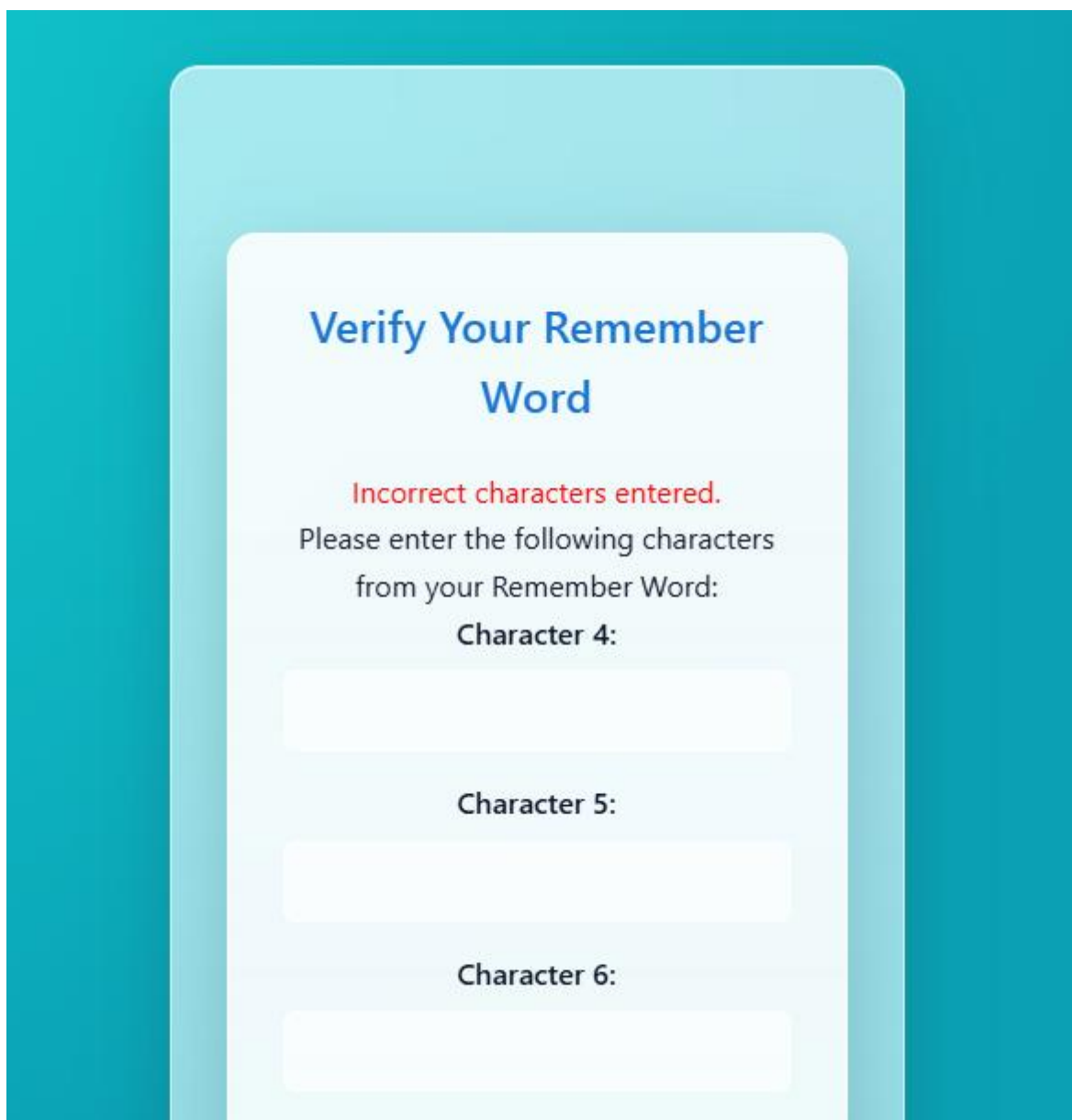
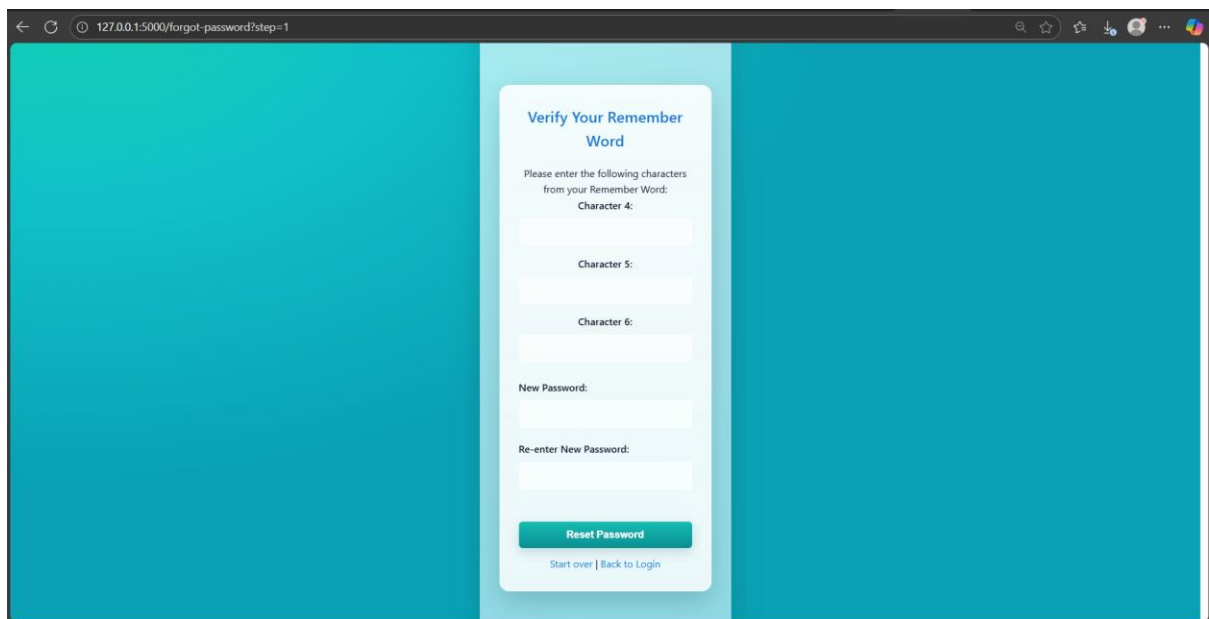
mp10

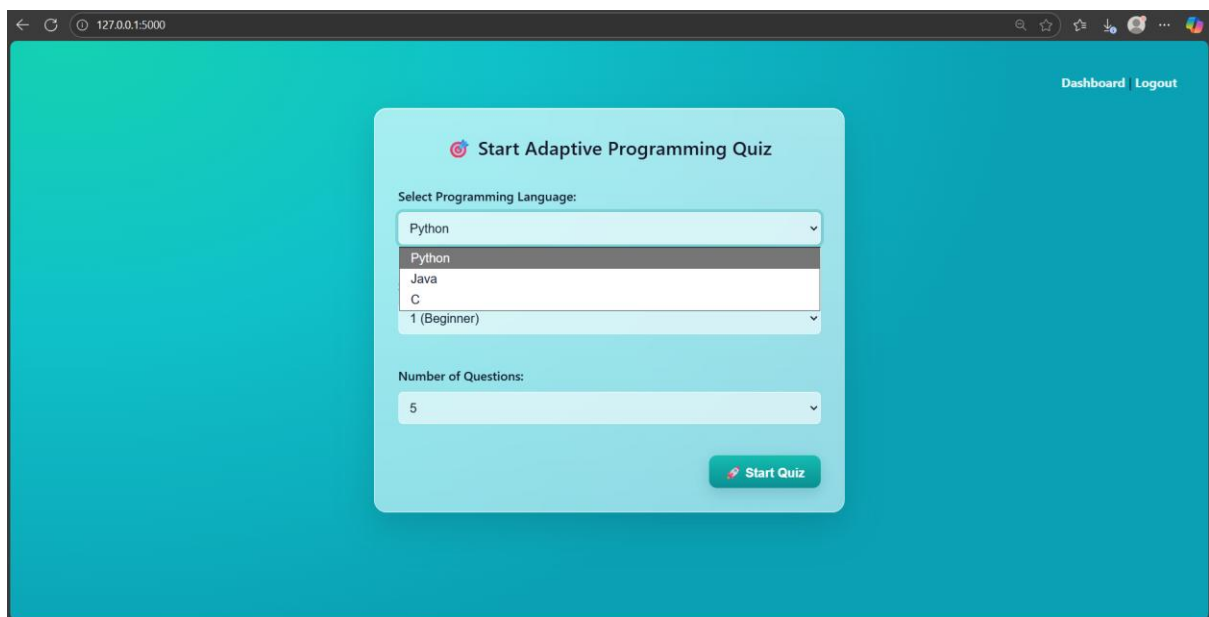
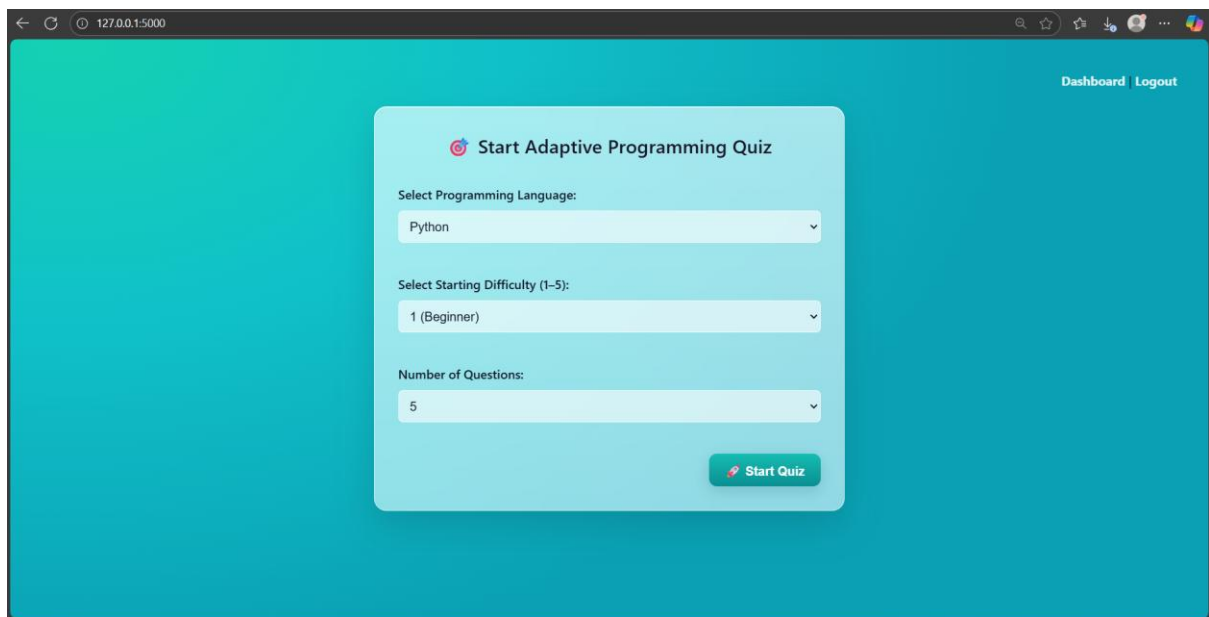
Passwords do not match.

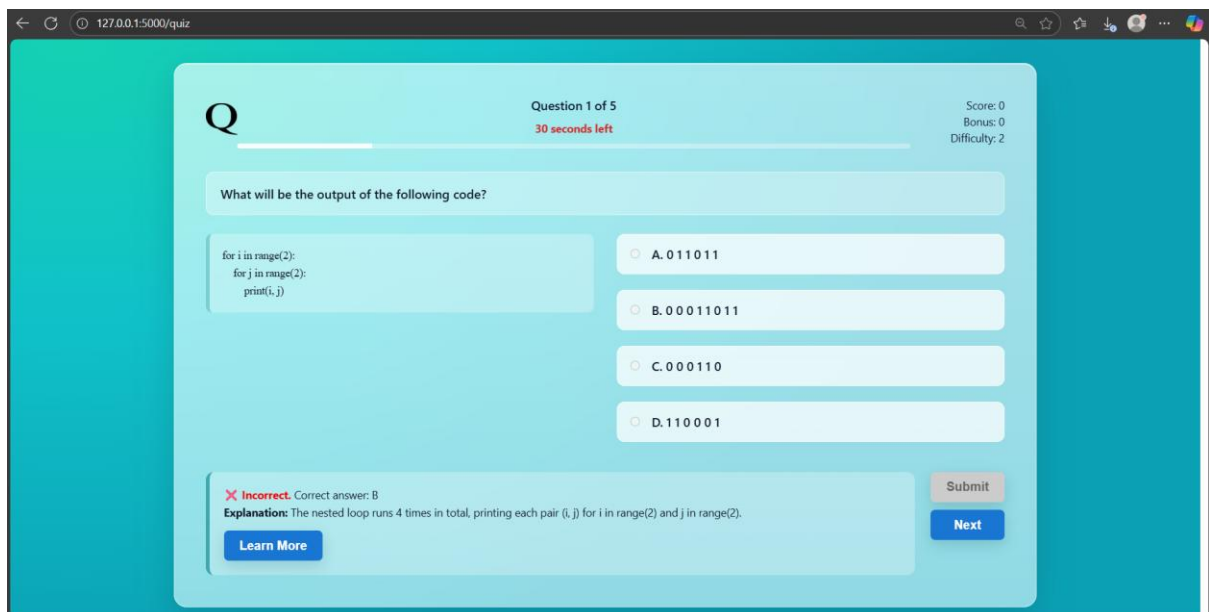
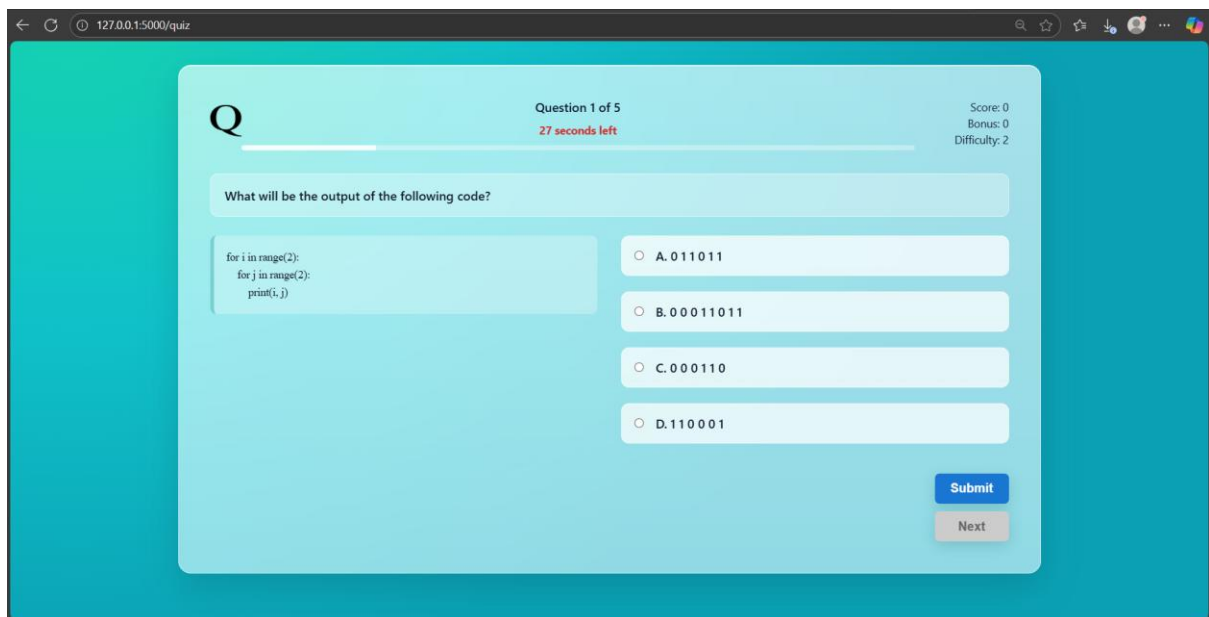
Sign Up

Already have an account? [Log in](#)









127.0.0.1:5000/quiz

```
for i in range(2):
    for j in range(2):
        print(i, j)
```

☐ A. 0 1 1 0 1 1

☐ B. 0 0 0 1 1 0 1 1

☐ C. 0 0 0 1 1 0

☐ D. 1 1 0 0 0 1

✖ Incorrect. Correct answer: B

Explanation: The nested loop runs 4 times in total, printing each pair (i, j) for i in range(2) and j in range(2).

Done

Your answer was close, but let's clarify. The code uses nested loops, where 'i' and 'j' both range from 0 to 1. This means for each value of 'i', 'j' will go through its full range. The output will be:

- 'i' = 0, 'j' = 0
- 'i' = 0, 'j' = 1
- 'i' = 1, 'j' = 0
- 'i' = 1, 'j' = 1

So, the correct sequence is '0 0\n0 1\n1 0\n1 1', which matches option B. Option A missed the initial '0 0' pair.

Submit

Next

127.0.0.1:5000/quiz?time_taken=0

Q

Question 3 of 5

30 seconds left

Score: 1
Bonus: 1
Difficulty: 2

What is the output of the following code?

```
for i in range(4):
    if i % 2 == 0:
        print(i)
```

☐ A. 0 1 2 3

☐ B. 0 2

☐ C. 1 3

☐ D. 2 3

✔ Correct!

Explanation: The loop prints the numbers that are even. Therefore, 0 and 2 are printed.

Learn More

Submit

Next

46

127.0.0.1:5000/quiz?time_taken=0

Question 4 of 5
30 seconds left

Score: 2
Bonus: 2
Difficulty: 2

Q

What is the output of this nested loop?

```
result = 0
for i in range(1, 4):
    for j in range(i):
        result += j
    print(result)
```

☐ A. 3

☐ B. 4

☐ C. 5

☐ D. 6

✓ Correct!

Explanation: The inner loop runs for each i, adding j to result. It adds 0 for i=1, 0+1 for i=2, and 0+1+2 for i=3, totaling 5.

Fetching...

Generating a deeper explanation...

Submit

Next

127.0.0.1:5000/quiz?time_taken=0

```
result = 0
for i in range(1, 4):
    for j in range(i):
        result += j
    print(result)
```

☐ A. 3

☐ B. 4

☐ C. 5

☐ D. 6

✓ Correct!

Explanation: The inner loop runs for each i, adding j to result. It adds 0 for i=1, 0+1 for i=2, and 0+1+2 for i=3, totaling 5.

Done

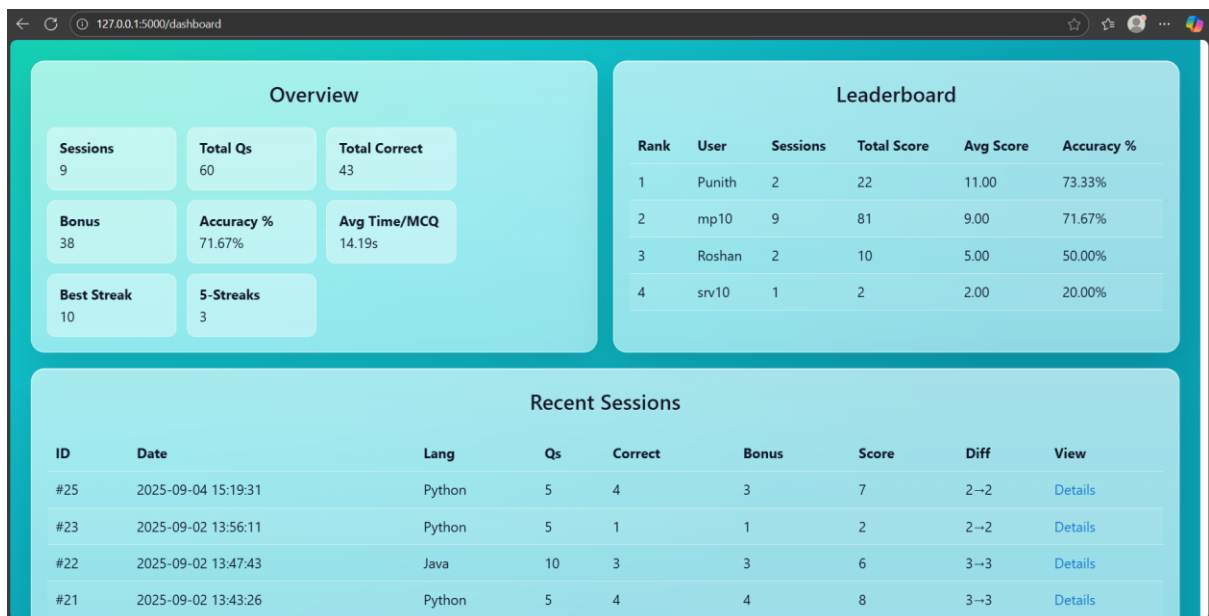
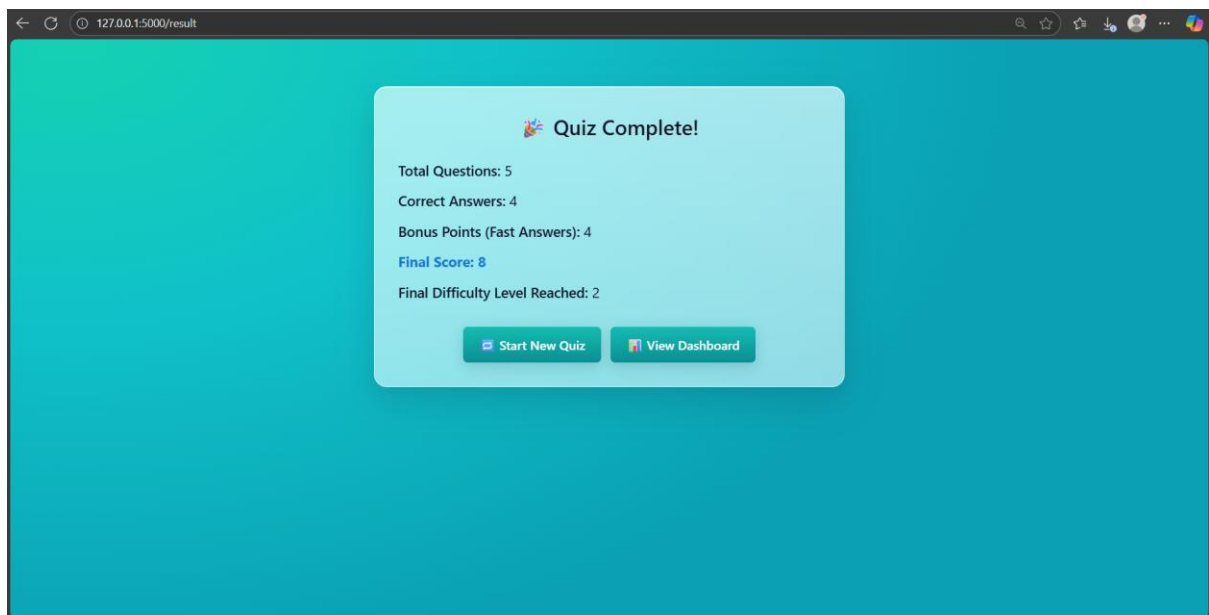
Great job! Your answer is correct. The code uses a nested loop where 'i' ranges from 1 to 3. For each 'i', the inner loop runs 'j' from 0 to 'i-1', adding 'j' to 'result'. Here's a quick breakdown:

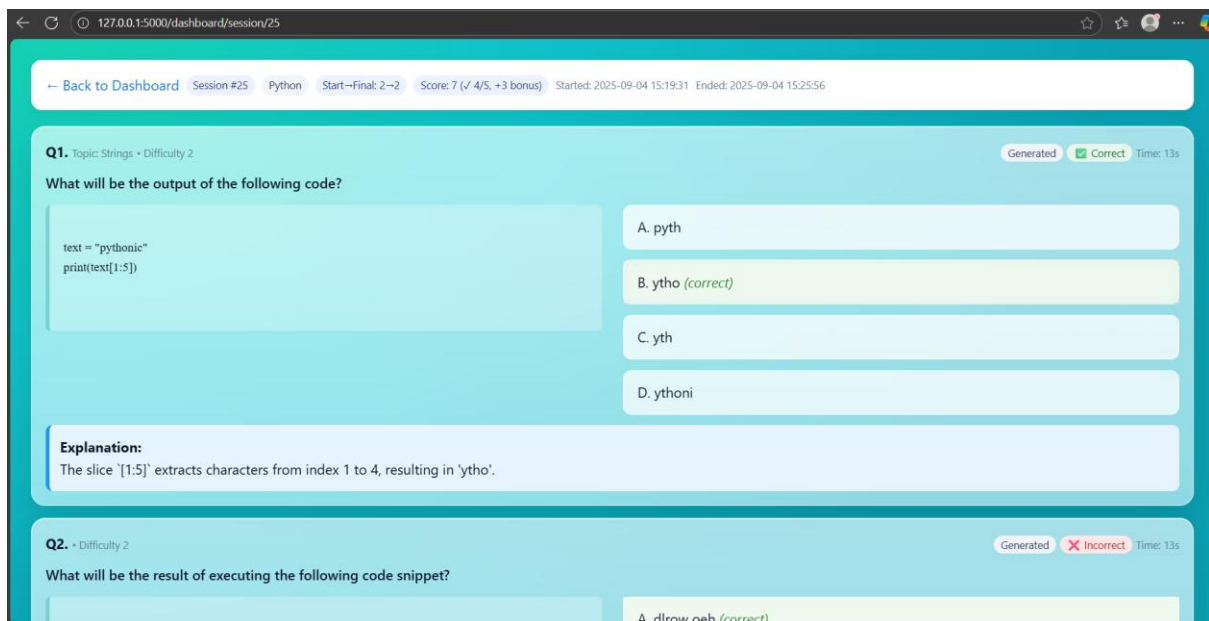
- When 'i' = 1, 'j' runs from 0 to 0, adding 0.
- When 'i' = 2, 'j' runs from 0 to 1, adding 0 and 1.
- When 'i' = 3, 'j' runs from 0 to 2, adding 0, 1, and 2.

Summing these gives 5. Options A, B, and D result from miscalculating the loops.

Submit

Next





B. MCQ JSON schema & dataset snippet

c.json object

```
{
  "id": "C2-LOOP-024",
  "topic": "For Loop",
  "difficulty": 2,
  "question": "What is the output?",
  "code": "#include <stdio.h>\nint main() {\n    for (int i = 0; i < 3; i++) {\n        printf(\"%d \", i);\n    }\n    return 0;\n}",
  "options": {
    "A": "0 1 2",
    "B": "1 2 3",
    "C": "0 1 2 3",
    "D": "0 1 2 "
  },
  "correct_option": "D",
  "explanation": "`printf()` adds a space after each number, so it prints: `0 1 2 ` (with trailing space).",
}
```

java.json object

```
{
  "id": "Q2-OPR-033",
```

```

"topic": "Operators",
"difficulty": 2,
"question": "What will be printed by this code?",
"code": "int x = 4;\nint y = 2;\nSystem.out.println(x > 3 || y < 1);",
"options": {
  "A": "true",
  "B": "false",
  "C": "1",
  "D": "Compile Error"},
"correct_option": "A",
"explanation": "`x > 3` is true, so the `||` (logical OR) short-circuits and returns true."
}

```

python.json	object
<pre> { "id": "PY2-CND-027", "topic": "Conditionals", "difficulty": 2, "question": "What is the output of this code?", "code": "x = 10\ny = 20\nif x > y:\n print(\"X\")\nelif x < y:\n print(\"Y\")\nelse:\n print(\"Equal\")", "options": { "A": "X", "B": "Y", "C": "Equal", "D": "None" }, "correct_option": "B", "explanation": "`x = 10`, `y = 20` → since 10 < 20, the `elif` block runs and prints 'Y'." } </pre>	

C. Diagrams (high-level arch, ER, state machine, pipeline)
D. API endpoints/IO contract
F. Prompt example

```

def higher_difficulty_prompt(example_json: dict) -> str:
    return (
        "You are writing a single-best-answer programming MCQ.\n"

```

```

"\n"
"GOAL\n"
"- Create a DIFFERENT question on the same general topic as the example, but with higher
difficulty via a subtle pitfall or deeper nuance.\n"
"- Keep it compact and unambiguous.\n"
"\n"
"HARD CONSTRAINTS (must pass all before you answer)\n"
"1) Exactly one option is correct; the other three are strictly incorrect under normal
interpretation.\n"
"2) Options are mutually exclusive: no logical equivalents, paraphrases, or overlapping truths.\n"
"3) Do NOT use 'All/None of the above' or combinations like 'A and C'. \n"
"4) Provide exactly four options with keys A, B, C, D.\n"
"5) If you include code, keep it minimal and self-contained (or use an empty string for code).\n"
"6) Double-check that correct_option matches one of A–D and that the chosen text actually
corresponds to the correct answer.\n"
"\n"
"QUALITY CHECKS (do before returning)\n"
"- Try to falsify each incorrect option; if any could be true in a normal reading, rewrite it.\n"
"- Imagine shuffling A–D: the same underlying text must remain the sole correct choice.\n"
"\n"
"Example MCQ JSON:\n"
f"{json.dumps(example_json, indent=2)}\n\n"
"Now produce a NEW, harder MCQ on a closely related subtopic.\n"
"Return ONLY valid JSON with this exact structure:\n"
"{\n"
'  "question": str,\n"
'  "code": str,          # "" if not needed\n"
'  "options": {"A": str, "B": str, "C": str, "D": str},\n"
'  "correct_option": "A"|"B"|"C"|"D",\n"
'  "explanation": str      # 1–3 concise sentences\n"
"}\n"
"Output JSON only."
)

```