International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 10 | Oct - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Placify AI: AI-Powered Placement Preparation Platform with Sandboxed Code Execution, Retrieval-Augmented Mentoring and Placement-Readiness Prediction

Mr. B Sanjay Krishna[1], Monish R[2], Nandha Kumar J[3], Pranesh I[4], Prasanna B[5], Rishi S[6]

[1] Assistant Professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: sanjaykrishnab@siet.ac.in

[2] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: monishr24cs@srishakthi.ac.in

[3] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: nandhakumarj24cs@srishakthi.ac.in

[4] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: praneshi24cs@srishakthi.ac.in

[5] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: prasannab24cs@srishakthi.ac.in

[6] Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
Email: rishis24cs@srishakthi.ac.in

ABSTRACT

Placify AI is a full-stack placement-preparation platform that brings a coding arena, structured learning tracks, mock interviews, resume analysis, contests and a community forum into one web application and adds machine-learning guidance on top. A React 19 single-page application talks to a TypeScript/Express API backed by SQLite through Prisma, while a separate Python FastAPI microservice hosts the intelligence layer: a monotone gradient-boosting placement-readiness model, a TF-IDF difficulty classifier and problem recommender, a sentence-embedding interview-answer scorer, and a retrieval-augmented mentor built on FAISS with optional Google Gemini generation. Untrusted submissions in five languages run in a locked-down Docker sandbox or Judge0. Authentication uses HS256 JSON Web Tokens with PBKDF2-SHA512 hashing, and every AI feature degrades to a documented deterministic fallback. On held-out data the placement model reaches 0.860 ROC-AUC and the difficulty classifier 76.7% accuracy; retrieval finds the expected knowledge chunk first for 83% of 30 queries; 103 ML tests and 60 sandbox checks pass. We also report weaknesses we measured: keyword-stuffed interview answers outscore partially correct ones, and the placement model is trained on synthetic data and is a demonstration only.

Keywords: placement preparation, online judge, retrieval-augmented generation, sentence embeddings, gradient boosting, FastAPI, React, Docker sandbox, FAISS, mock interview, educational technology

I. INTRODUCTION

Campus placement is a decisive milestone for engineering students, yet preparation is usually scattered across unrelated tools: one site for coding problems, another for aptitude or theory notes, a third for mock interviews, and a spreadsheet to track progress. Students rarely receive a single, honest answer to the question that matters most: *am I ready, and what should I do next?* Institutions face the mirror-image problem, since they have little visibility into how a cohort is actually preparing, and mentors cannot scale one-to-one guidance.

Placify AI addresses this by combining the main preparation activities in one application. The *Arena* offers 276 coding problems with sample and hidden test cases, an in-browser editor, and execution in five languages. *Learning Tracks* cover 9 syllabi with 589 topics and quizzes. *Mock Interviews* present technical, HR and behavioural questions and score the answers. The platform also provides a resume analyser, a Placement Hub of revision notes, flashcards and multiple-choice questions, career roadmaps, company-wise problem lists, timed contests, a discussion forum, an admin studio, and a gamification layer of XP, levels, streaks and badges that rewards consistent practice.

The central contribution is the way machine learning is integrated without compromising reliability. All intelligence lives in a separate Python FastAPI [4] service that the Node.js server reaches through a shared-secret header; if that service is unreachable, the web application keeps working and clearly labels the heuristic fallbacks it uses instead. Running arbitrary student code is the most security-critical feature of such a platform, so submissions never execute inside the API process: they run in one locked-down container per submission or on a hosted Judge0 instance.

The objectives of the work are:

- to design a single platform that unifies practice, learning, interview rehearsal and progress analytics;
- to execute untrusted code in five languages safely, without leaking hidden test data;
- to provide transparent, app-relative readiness scoring with actionable next steps;
- to ground an AI mentor in the platform's own knowledge base and keep every AI feature optional; and
- to evaluate each component with measurements, including its weaknesses.

The main contributions are (i) a modular architecture that separates transactional and AI workloads; (ii) a monotone, app-relative placement-readiness model with a documented fallback; (iii) a sandboxed multi-language judge validated by 60 isolation and verdict checks; (iv) empirical evaluations of the placement, difficulty, recommendation, retrieval and interview-scoring components, including permutation importance, calibration, retrieval hit rates and an ablation that exposes a keyword-stuffing weakness; and (v) an end-to-end latency and throughput benchmark of the running system.

Section II surveys related work, Section III states requirements, Section IV presents the methodology, Section V the implementation, Section VI the advantages, Section VII the

experiments, Section VIII limitations and ethics, and Sections IX and X conclude and outline future work.

II. LITERATURE SURVEY

A. Online judges and programming education

Online judge systems automatically compile and run submitted programs against test data and return verdicts. Early work such as Kurnia et al. [21] showed that automated judging supports programming courses, and the survey of Wasik et al. [20] catalogues the architecture, security and evaluation issues of such systems. Their recurring concerns are safe execution, fair resource limits, and protection of hidden tests. Commercial practice sites popularised large problem banks, company-tagged question lists and mock interviews, but practice, learning material, interview rehearsal and analytics typically sit in separate products, so no component can use evidence from the others.

B. Sandboxing untrusted code

Container isolation with dropped capabilities, resource limits and network denial is the common approach for self-hosted judges [16], [22], while open-source systems such as Judge0 [17] expose sandboxed execution as a service. Containers share the host kernel, so defence in depth (no network, read-only root file system, process and memory limits, non-root user) matters. Placify AI supports a Docker runner, Judge0 and a development-only local runner and selects the back-end through configuration.

C. Gamification and learner engagement

Gamification applies game-design elements outside games [23]. Empirical reviews report mostly positive but context-dependent effects on engagement [24], which motivates XP, levels, streaks and badges in Placify AI while keeping rewards tied to verified outcomes: XP is granted only for the first accepted solution of a problem and for completed interviews and topics.

D. Educational data mining and placement prediction

Educational data mining applies statistical and machine-learning methods to learner data [25]. Predicting employability or placement from academic and activity records is a common application, but real outcome data are institution-specific, small and sensitive. Placify AI therefore separates the *modelling pipeline* from the *data*: it ships an app-relative feature design, a monotone model and an evaluation protocol that work with synthetic data today and can be re-fitted on consented outcomes later.

Gradient-boosted trees [6] are strong on tabular data, and monotonic constraints let a model encode the domain rule that more practice must never lower a readiness estimate [32]. Probability quality is assessed with the Brier score [29] and calibration analysis [30]; permutation importance [31] gives a model-agnostic view of which features matter.

E. Text representations, recommendation and semantic scoring

TF-IDF term weighting [7] with naive Bayes [8] or linear classifiers remains a strong, interpretable baseline for small labelled corpora, and cosine similarity between a user profile and item vectors yields a simple content-based recommender. Sentence-BERT [9] and distilled encoders such as MiniLM [10] produce compact sentence embeddings; BERT-style pre-training [26] underlies them. Combined with an inner-product index such as FAISS [11] they enable low-latency dense retrieval [27], and similarity to reference answers is a practical way to score free-text answers without labelled training data.

F. Retrieval-augmented generation

Large language models can produce fluent but unsupported statements [28]. Retrieval-augmented generation [12] conditions the model on retrieved passages to ground its answers. Placify AI follows this pattern for its mentor, retrieving chunks of its own knowledge base and prompting Gemini [18], [33] with them, and falls back to extractive synthesis from the same chunks when generation is unavailable.

G. Research gap

Prior work treats judging, recommendation, answer scoring and retrieval as separate problems and rarely reports component-level weaknesses. We contribute an integrated, open design in which every AI component is optional, labelled and evaluated, together with a service-oriented separation [19] that lets the intelligence layer evolve independently of the transactional core.

III. REQUIREMENTS AND DESIGN GOALS

Requirements were derived from the needs of students (practice, feedback, motivation), administrators (content and user management) and operators (safety, portability). Functional requirements are listed in TABLE I.

TABLE I. FUNCTIONAL REQUIREMENTS

ID	Requirement
FR1	Register and log in by e-mail or username; JWT-secured sessions.
FR2	Browse, search and filter problems by difficulty, topic and company tag.
FR3	Run (visible tests) and submit (all tests) code in JavaScript, Python, Java, C++ and C.
FR4	Award XP, levels, streaks and badges for verified activity.
FR5	Study learning-track topics with quizzes; track completion.
FR6	Take Technical, HR and Behavioural mock interviews with scored answers.
FR7	Analyse a resume against a target role and return an ATS-style score.
FR8	Ask an AI mentor questions grounded in the knowledge base.
FR9	Register for timed contests; discuss and reply in a community forum.
FR10	Show readiness score, next-step insights, recommendations and activity analytics.
FR11	Let administrators manage problems, contests, users and moderation.

Non-functional goals shaped the design: *security* (untrusted code, hidden tests, credentials); *graceful degradation* (the ML service and Gemini are optional); *honesty* (labelled fallbacks, no fabricated scores); *portability* (a single SQLite file, no external services required, Windows and Linux); *maintainability* (one router per feature, shared validation and error handling); and *testability* (the API, the ML service and the sandbox each have automated suites).

Three design decisions deserve explanation. First, SQLite [34] with Prisma [3] keeps setup to a single command and suits a campus-scale deployment; the schema is expressed in Prisma, so moving to a server database is a configuration change. Second, the intelligence layer is a separate FastAPI service rather than in-process code, because it needs a different runtime (Python, PyTorch, FAISS) and a different release cadence. Third, every AI feature has a deterministic fallback so that an outage degrades quality, not availability.

TABLE II. DESIGN ALTERNATIVES AND CHOICES

Decision	Chosen	Alternative and trade-off
Database	SQLite + Prisma	PostgreSQL: better concurrency, needs a server
AI placement	In a separate FastAPI service	In-process: simpler, but no Python ML stack
Readiness model	Monotone gradient boosting	Logistic regression: slightly higher AUC, not monotone

Decision	Chosen	Alternative and trade-off
Recommender	TF-IDF content-based	Embeddings or collaborative filtering: need more data
Interview scoring	Embedding similarity + keywords	LLM judge: richer, but costly and non-deterministic
Mentor	RAG with local fallback	LLM only: fluent but ungrounded
Code runner	Docker per submission	Judge0: no host Docker, external dependency

IV. PROPOSED METHODOLOGY

The methodology rests on four principles: *evidence-based guidance* (every score is computed from measurable activity), *safe execution* (untrusted code is isolated and answers are never leaked), *graceful degradation* (each AI feature has a deterministic fallback), and *modular scalability* (the ML service evolves independently of the transactional API).

A. Architecture

A request is authenticated by the Express server, validated, processed against SQLite, and, where intelligence is needed, forwarded to the ML service whose result is returned and, where appropriate, persisted (Fig. 1). The system comprises authentication, problem arena and code execution, learning tracks, mock interviews, resume analysis, contests, discussions, dashboard analytics, an AI mentor and administration; each is a separate Express router with schema-validated input.

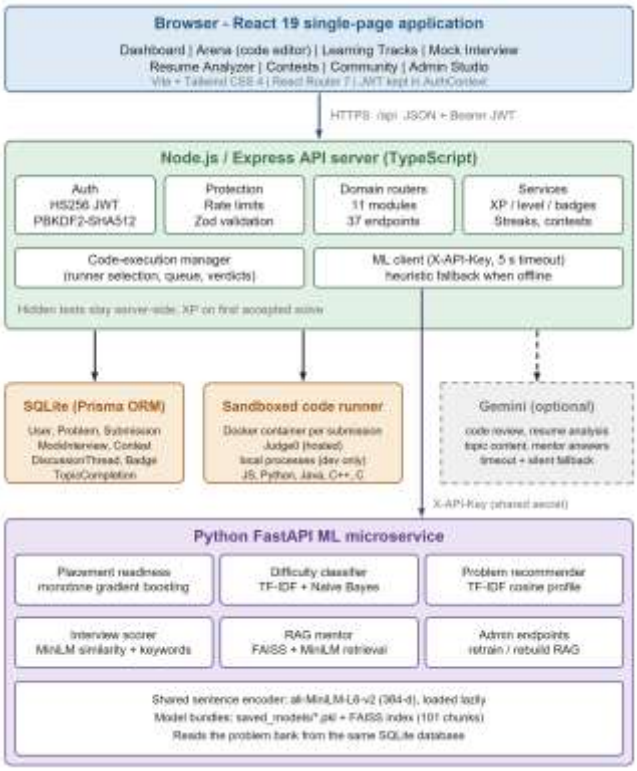


Fig. 1. High-level architecture of the Placify AI platform.

B. Code-submission workflow

Fig. 2 shows the path of a submission. After authentication, validation and rate limiting, the server loads the problem with all of its tests, but sends only the source code and the test *inputs* to the sandbox; expected outputs never enter the container. The driver compiles once, runs each test in a fresh process, and returns raw outputs, which the server compares while ignoring trailing whitespace. A *Run* executes the visible tests only and stores nothing; a *Submit* executes all tests, stores the result, and grants XP on the first accepted solution. Failures on hidden tests reveal no input, output or error details.

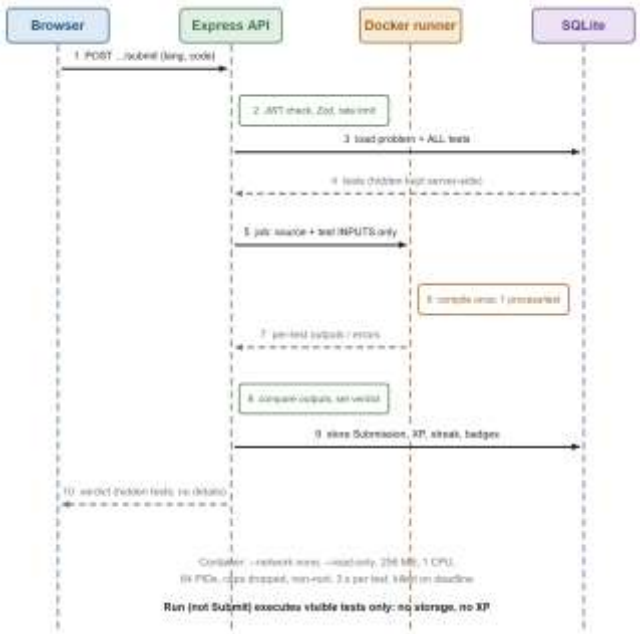


Fig. 2. Sequence of a code submission.

Algorithm 1. Judging a submission

```
input: user u, problem q, language L, code C, mode m
1  validate token, schema(L, C), rate limit; else reject
2  tests <- visible(q) if m = Run else all(q)
3  job <- (C, L, inputs(tests))  // outputs stay server-side
4  out <- sandbox.run(job)    // 1 container, 3 s per test
5  verdict <- compare(out, expected(tests))
6  if m = Run: return verdict; store nothing
7  begin transaction
8    store Submission(u, q, L, C, verdict)
9    if Accepted and q unsolved by u: grant XP(difficulty)
10   update streak, accuracy, badges
11 commit; return verdict, hiding hidden-test detail
```

C. Placement-readiness model

Readiness is measured relative to the application itself: solving 20 problems means a lot when the bank holds 21 and little when it holds 2,000. Raw activity is therefore converted into seven app-relative features (TABLE III) by a single function shared by training and inference, so the two cannot drift apart. Raw XP and level are deliberately excluded because their scale grows with the size of the bank.

TABLE III. READINESS FEATURES AND SIGNAL WEIGHTS

Feature	Definition	Weight
solved_ratio	problems solved / bank size; full credit at 80%	0.35
accuracy	accepted / graded submissions, scaled by evidence	0.20
interview_average	mock-interview mean (30 to 0, 85 to 1); has_interview flag	0.15
streak_sig	min(streak, 30) / 30	0.10
topic_ratio	topics completed / curriculum; full credit at 25%	0.10
practice_volume	submissions per bank problem (cap 3; full at 1.5)	0.10

Each component c is normalised to [0, 1] and the readiness signal s is their weighted sum, with the weights of TABLE III:

$$s = 0.35\,c_{solved} + 0.20\,c_{acc} + 0.15\,c_{int} + 0.10\,c_{streak} + 0.10\,c_{topic} + 0.10\,c_{prac} \quad (1)$$

where c_solved = min(1, solved_ratio / 0.8), c_acc = clip((accuracy - 30) / 60) * min(1, practice_volume / 0.25) and c_int = clip((average - 30) / 55) when an interview exists and 0 otherwise. Because no real placement outcomes exist, the model is trained on a synthetic cohort of 8,000 users (class-balanced) generated from a latent engagement variable u ~ Beta(1.1, 2.2). The label follows a noisy rule:

$$y = 1 \text{ if } s + e >= 0.62, \text{ with } e \sim Logistic(0, 0.13) \quad (2)$$

TABLE IV. SYNTHETIC COHORT USED TO TRAIN THE READINESS MODEL

Quantity	Generating distribution
Users	8% new accounts; otherwise latent u ~ Beta(1.1, 2.2)
Bank / curriculum size	Uniform: 15-150 problems, 40-600 topics
Solved share	1.05 u + N(0, 0.08), clipped to [0, 1]
Accepted / failed submissions	solved + Poisson(0.2 solved); Poisson((1.6 - 1.4u) solved + <= 4% of bank) x LogNormal(0, 0.4)
Streak	0 with probability 0.55 - 0.45u, else Exponential(mean 2 + 25u) days
Mock interview	taken with probability 0.1 + 0.7u; average ~ N(35 + 50u, 12)
Topic share	Beta(0.8, 10) x (0.3 + 2u)
Cohort size	22,897 raw users (17.5% ready), down-sampled to 4,000 + 4,000

Candidates are logistic regression and histogram gradient boosting with monotonic constraints [5], [6]; a candidate is rejected if a probe shows its score ever drops when a monotone feature rises. Candidates are ranked by 5-fold cross-validated ROC-AUC on the training split only, and the winner is evaluated once on a held-out 20% test split. The reported score and the documented fallback are:

$$score = round(100\ p); \quad p_fallback = 1 / (1 + exp(-(s - 0.42) / 0.13)) \quad (3)$$

Every response carries is_demo = true. Up to four insights, ranked by the readiness gain each action would produce (for example, solving enough problems to reach the next 25%, 50% or 80% milestone of the bank), tell the student what to do next.

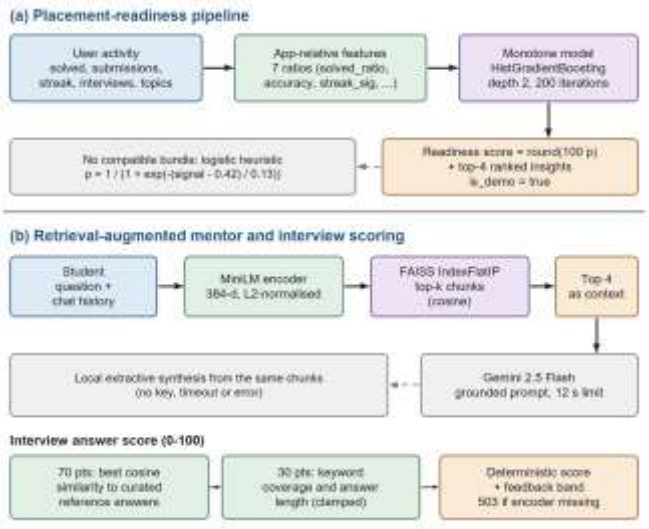


Fig. 3. (a) Placement-readiness pipeline; (b) RAG mentor and interview-answer scoring.

Algorithm 2. Readiness score and insights

```
input: activity counts and bank sizes
f <- engineer_features(counts)  // 7 ratios
if compatible trained bundle exists:
      p <- model.predict_proba(f)  // monotone GBM
else p <- 1 / (1 + exp(-(s(f) - 0.42) / 0.13))
score <- round(100 p);  ready <- (p >= 0.5)
for each action a in the five signal components:
      gain(a) <- weight(a) x (c_after - c_now)
insights <- top 4 actions by gain(a) > 0
return score, ready, insights, model, is_demo = true
```

D. Difficulty classifier and recommender

The difficulty classifier maps a problem's title, tags, description and constraints to Easy, Medium or Hard. It is trained on 297 curated, hand-labelled examples (99 per class). Logistic regression, a calibrated linear SVM and multinomial naive Bayes, all on TF-IDF features (up to 1,500 terms, 1-3-grams), are compared by 5-fold cross-validation; the predicted probabilities always sum to one.

The recommender represents each problem as an L2-normalised TF-IDF vector v of its title, tags (repeated so that topic overlap outweighs incidental words) and description. A user profile is the mean vector of the solved set S, and unsolved problems are ranked by cosine similarity using sparse products:

$$u = (1/|S|) \sum_{i \in S} v_i; \quad r_j = (v_j \cdot u) / ||u|| \quad (4)$$

For users with no history the recommender falls back to easiest-first ordering.

E. Interview-answer scorer

An answer is scored from 0 to 100 without any training step. Let e_a be the MiniLM embedding of the answer and e_r the embeddings of the curated reference answers matched to the question; n is the word count, U the set of distinct words and W the technical keyword list:

$$score = 70 \max_r \cos(e_a, e_r) + 30 [0.5 \min(1, 10 |U \& W| / |U|) + 0.5 \min(1, n / 80)] \quad (5)$$

Answers under three words score zero, and if the encoder cannot be loaded the API answers with HTTP 503 rather than returning a fabricated score. A reference set is chosen by matching question words against reference keys; otherwise a generic reference is used. Section VII-F shows why this matching matters.

F. Retrieval-augmented mentor

The knowledge base consists of 101 chunks drawn from problem statements, DSA notes, a Python track, company preparation, system design and interview tips; community discussions are deliberately not indexed because they are unmoderated. Chunks are encoded with all-MiniLM-L6-v2 (384 dimensions), L2-normalised and stored in a FAISS flat inner-product index, which equals cosine similarity. For each question the top four chunks and up to ten turns of chat history form a grounded prompt for Gemini 2.5 Flash with a 12-second limit; on a missing key, timeout or error the answer is synthesised extractively from the same chunks. The response reports its method and the sources used.

TABLE V. KNOWLEDGE-BASE SOURCES

Source	Chunks	Content
Problem Bank	60	Title, difficulty, tags, description, hints, editorial
DSA	15	Arrays to bit manipulation: patterns and complexities
Python Track	11	Language fundamentals to asyncio and ML basics
Company Prep	6	TCS, Infosys, Wipro, Google, Amazon, Microsoft
System Design	5	Caching, load balancing, sharding, APIs, queues
Interview Tips	4	Strategy, HR, system design, resume and ATS

Algorithm 3. Mentor answer

```
input: question q, chat history H (last 10 turns), top_k = 5
e <- normalize(encoder(q)); R <- index.search(e, 5)
if R is empty: return 'not found' (no-retrieval)
if GEMINI_API_KEY set:
      prompt <- instructions + top 4 chunks of R + H + q
      a <- Gemini(prompt) within 12 s
      if a: return a, sources(R), method = 'RAG + Gemini'
a <- extractive_summary(top 3 chunks of R)
return a, sources(R), method = 'RAG (local synthesis)'
```

G. Resume analyser

The resume analyser returns an ATS-style score with strengths, improvements and matched or missing keywords. With a Gemini key it requests structured JSON output and treats the resume strictly as data between explicit markers; otherwise a deterministic heuristic is used. The heuristic selects one of seven role profiles (for example Frontend, Backend, Full Stack, Data Science, DevOps) from the target role and awards up to 40 points for keyword coverage (with aliases such as "sklearn" for Scikit-learn), 20 for standard sections, 15 for quantified achievements, 10 for action verbs, 10 for contact and profile links and 5 for length.

H. Gamification model

XP is 20, 50 or 100 for the first accepted Easy, Medium or Hard solution, 30 for a completed mock interview and 15 for a completed learning topic. The level is floor(XP / 500) + 1. A daily streak is maintained from UTC activity dates, and fifteen milestone badges are awarded idempotently.

V. SYSTEM IMPLEMENTATION

A. Frontend

The client is a React 19 [1] single-page application built with Vite, Tailwind CSS 4 and React Router 7, written in TypeScript. Shared layout, theme and authentication contexts keep navigation, light and dark themes and the JWT session consistent across pages. Route guards separate student and administrator views, an error boundary isolates rendering failures, markdown is rendered lazily, and a command palette provides keyboard navigation. TABLE VI lists the pages.

TABLE VI. APPLICATION PAGES

Page	Purpose
Login	Registration and login by e-mail or username.
Dashboard	Readiness hero, insights, recommendations, strong/weak topics, 84-day activity heatmap.
Arena	Problem list with filters; editor, Run/Submit, results and AI review.
Learning Tracks	Nine syllabi; topic pages with theory, examples, challenges and quizzes.
Mock Interview	Technical, HR and Behavioural sessions with per-answer feedback.
Resume Analyzer	ATS-style score for a target role.
Contests	Timed contests; registration.
Community	Threads, replies, likes.
Company Prep	Problems grouped by company tag.
Placement Hub	Notes, flashcards and MCQs for OS, DBMS, networks, SQL and aptitude.
Career Roadmap	Five role paths (e.g. FAANG SDE, Frontend, Fullstack, Data, Systems) with milestones.
Profile	XP, level, streak, badges, submission history.
Admin Studio	Overview statistics; problem, contest, user and discussion management.

Fig. 4 to Fig. 7 show the running application. The Dashboard opens with a readiness hero ("17% ready" for the seeded demo student, with the banner "Demo model trained on synthetic data, not a hiring prediction"), a three-step onboarding strip, recommended problems and ranked insights. The Arena shows the problem statement, constraints, formats and examples beside the editor, with Run and Submit.



Fig. 4. Dashboard: readiness hero, onboarding, recommendations and insights.



Fig. 5. Arena: problem statement and code editor with Run / Submit.



Fig. 6. Mock Interview set-up: Technical, HR and Behavioural sessions.

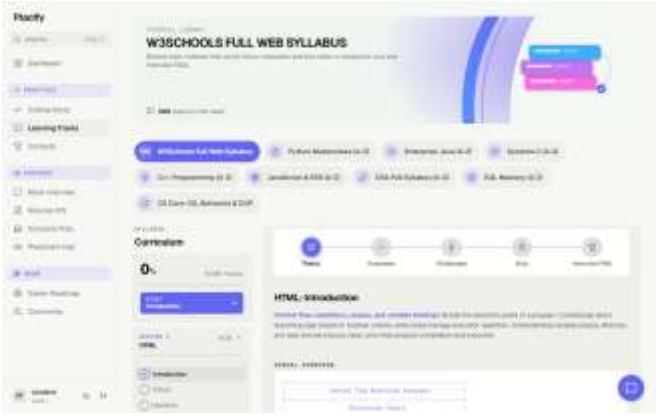


Fig. 7. Learning Tracks: nine syllabi with a five-stage topic page.



Fig. 8. Resume Analyzer: role-specific ATS-style score with strengths and improvements.

Without a Gemini key, a topic page shows static content that is generic to the track rather than specific to the topic (the HTML "Introduction" page in Fig. 7 discusses control flow in general terms), so the learning tracks are only as rich as the optional AI generation allows.

B. Backend and API

The server uses Node.js 20+, Express [2] and TypeScript with 11 route modules exposing 37 endpoints (TABLE VII). Request bodies are limited to 100 KB, cross-origin access is restricted to configured origins, an unknown /api route returns a JSON 404 rather than the SPA, and a final error handler hides internal details in production.

TABLE VII. API MODULES UNDER /api

Module	Endpoints	Access
auth	POST register, login, logout; GET me	public / JWT
problems	GET list, GET :id, POST :id/submit; admin POST, PUT, DELETE, GET :id/full	public / JWT / admin
submissions	GET own history	JWT
dashboard	GET analytics (readiness, recommendations, activity)	JWT
mentor	POST ask (RAG)	JWT, AI limit
mock-interview	POST start, POST :id/answer	JWT, AI limit
learning-tracks	GET list, GET topic, POST topic complete	public / JWT
resume	POST analyze	JWT, AI limit
contests	GET list, POST :id/register	optional JWT / JWT
discussions	GET list, POST thread, reply, like	JWT
admin	stats; users (list, patch, reset, delete); contests; moderation	admin
health	GET runner and ML status	public

The ML service exposes the endpoints of TABLE VIII; all but /health require the shared X-API-Key header, and Node calls them with timeouts of 5 s (10 s for scoring, 20 s for the mentor).

TABLE VIII. ML SERVICE ENDPOINTS

Endpoint	Purpose
GET /health	Which models and the RAG index are loaded (public)
POST /ml/placement-score	Readiness score, probability, insights, model name, is_demo
POST /ml/recommend-problems	Ranked unsolved problems for a solved set
POST /ml/difficulty-predict	Easy/Medium/Hard with class probabilities
POST /ml/interview-score	Answer score, feedback, semantic similarity
POST /rag/mentor-ask	Grounded answer, sources, method
POST /rag/retrieve	Top-k chunks with scores
POST /admin/retrain, /admin/rebuild-rag	Retrain all models / rebuild the index and reload

C. Data model

Prisma maps ten models onto SQLite (Fig. 9), with indexes on submissions by user and time and by problem; child rows are removed with their parent by cascading deletes. Array-valued fields such as tags, examples and test cases are stored as JSON text because SQLite has no native array type. Per-user sequences such as interview answers and XP updates run under a per-user lock, and interview answers use an optimistic concurrency check so that two simultaneous answers cannot corrupt the session.

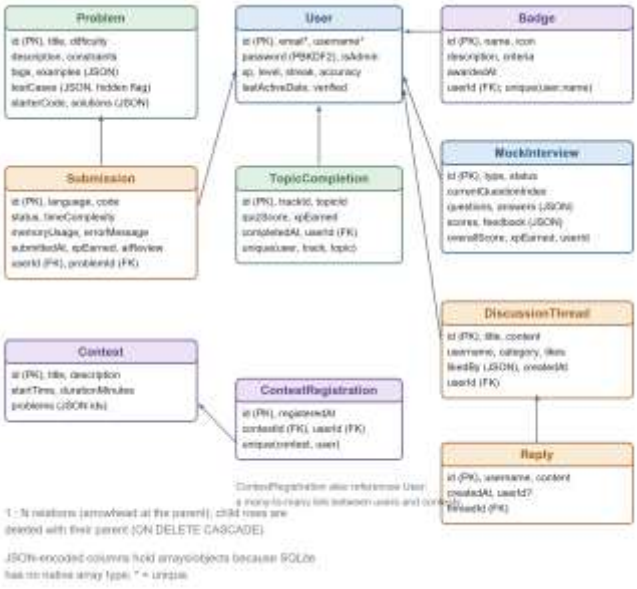


Fig. 9. Entity-relationship diagram of the ten Prisma models.

D. Security

Authentication uses stateless HS256 JSON Web Tokens [13] whose header algorithm is checked and whose signature is compared in constant time. Passwords are hashed with PBKDF2-HMAC-SHA512 at 210,000 iterations and a random salt [14], [35]; legacy hashes are verified and upgraded at the next login. Login and registration are limited to 20 requests per minute per address, and code-execution and AI routes to 30. The server refuses to start without a JWT secret of at least 32 characters. TABLE IX summarises the threat model in the spirit of the OWASP API guidance [15].

TABLE IX. THREAT MODEL

Threat	Mitigation	Evidence
Sandbox escape	Container: no network, read-only root, non-root, caps dropped, no-new-privileges	Runner tests
Resource exhaustion	256 MB, 1 CPU, 64 PIDs, 3 s per test, 64 KB output, 4 concurrent containers, server deadline kill	Runner tests
Hidden-test leakage	Expected outputs stay server-side; hidden failures show no details	API tests
Token forgery	Algorithm check, constant-time HMAC compare	API test
Brute force	PBKDF2 210k iterations; 20 requests/min	API tests
Privilege escalation	Admin flag ignored at registration; admin middleware	API tests
Malformed input	Zod schemas; 100 KB body limit; parameterised queries	API tests
Prompt injection (resume)	Resume delimited and treated as data	Design only

Threat	Mitigation	Evidence
Unmoderated text in RAG	Discussions not indexed	Design only
Solution disclosure	Residual: problem endpoints return reference solutions	By design

E. Sandboxed code execution

The solution contract is uniform across languages: the program receives the test input as a string and returns the output string. TABLE X compares the three runners. In Docker mode the driver is PID 1 inside the container, is non-dumpable so user code cannot kill it or forge its result, compiles once (javac, g++ or gcc), runs each test in a fresh process, and zeroes IPC limits so nothing survives between tests. A /tmp tmpfs of 64 MB is wiped between tests, and no server environment variable is passed in.

TABLE X. CODE-RUNNER BACK-ENDS

Runner	Languages	Isolation
docker (production)	JS, Python, Java 21, C++17, C11	Per-submission container: no network, read-only root, 256 MB, 1 CPU, 64 PIDs, all capabilities dropped, non-root user
judge0 (production)	JS, Python, Java, C++, C	Judge0 isolate sandbox with CPU, wall-time and memory limits
local (development)	JS, Python	Separate processes with language-level guards; not a hardened sandbox

F. Problem bank

The Arena contains 276 problems (158 Easy, 102 Medium, 16 Hard) in 20 topic categories (TABLE XI) with 2,221 test cases, of which 1,669 (75%) are hidden. Each problem has constraints, input and output formats, examples, hints, an editorial, starter code in five languages and reference solutions. Company tags come from curated lists where available and from a deterministic pseudo-random fallback elsewhere, so they are illustrative and are not interview-frequency data.

TABLE XI. PROBLEM-BANK COMPOSITION BY TOPIC

Topic	n	Topic	n
Bit Manipulation	21	Heaps	14
Advanced Interview	20	Segment Trees	14
Graphs	19	Greedy	14
Dynamic Programming	19	Queues	13
Arrays	17	Two Pointers	13
Hashing	16	Sliding Window	12
Recursion	15	Tries	10
Strings	14	Stacks	6
Linked Lists	14	Backtracking	6
Trees	14	Binary Search Trees	5

G. Learning tracks

Nine tracks hold 589 topics (TABLE XII). Topic pages are generated on demand by Gemini with a strict JSON schema (theory, diagram, code examples, practice questions, challenges, five quiz questions and interview questions), validated with Zod and cached; without a key the page falls back to static content. Completing a topic records the quiz score and awards XP once.

TABLE XII. LEARNING TRACKS

Track	Topics
W3Schools full web syllabus	286
DSA full syllabus (A-Z)	63
SQL mastery (A-Z)	55
Python masterclass (A-Z)	40
CS core: OS, networks and OOP	39
Enterprise Java (A-Z)	33
Systems C (A-Z)	27
JavaScript and ES6 (A-Z)	26
C++ programming (A-Z)	20

H. Gamification, contests, community and administration

TABLE XIII lists the XP rules and the fifteen badges. Contest registration is idempotent. Discussion threads support replies and likes, and posting earns a badge. The Admin Studio provides overview statistics and CRUD for problems (with an admin-only endpoint that returns hidden tests), contests, users (promote, reset progress, delete) and moderation of threads and replies.

TABLE XIII. XP RULES AND BADGES

Rule / badge	Criterion
XP: problem	First accepted solve: Easy 20, Medium 50, Hard 100
XP: interview / topic	Completed interview 30; completed topic 15
Level	floor(XP / 500) + 1
First Solve, Problem Solver, Problem Crusher, Code Master	1, 10, 25, 50 problems solved
Consistency Streak, Fortnight Focus	7- and 14-day streaks
Interview Starter, Interview Pro	1 and 5 completed interviews
Curious Learner, Knowledge Seeker	1 and 10 completed topics
Contest Entrant, Community Voice	First contest registration; first post
Rising Star, XP Hunter	Level 5; 1,000 XP
Sharpshooter	5 problems solved at >= 80% accuracy

I. Configuration and deployment

A setup script creates the environment file with random secrets, applies the schema and seeds the bank; seeding is idempotent. A single batch file starts the ML service (creating its virtual environment and training missing models) and the web server, and checks that the configured ports are free. A production build bundles the client and a single server file; in production demo accounts are not seeded and the local runner is disabled unless selected explicitly. TABLE XIV lists the main settings.

TABLE XIV. MAIN CONFIGURATION SETTINGS

Variable	Effect
JWT_SECRET, INTERNAL_API_KEY	Required secrets (>= 32 characters / shared with ML service).
CODE_RUNNER	docker, judge0 or local; empty selects Judge0 if configured, else local outside production.
ML_SERVICE_URL, ML_PORT	Where Node reaches the ML service; its bind port.
GEMINI_API_KEY, GEMINI_MODEL	Optional generation (default gemini-2.5-flash).
RATE_LIMIT_AUTH_MAX, RATE_LIMIT_AI_MAX	Requests per minute per address (20 / 30).
ALLOWED_ORIGINS	Origins permitted for cross-origin calls.

TABLE XV. TECHNOLOGY STACK

Layer	Technology (version)
Client	React 19, React Router 7, Vite 6, Tailwind CSS 4.1, TypeScript 5.8, Motion, lucide-react
Server	Node.js 20+ (tested on 24), Express 4.21, express-rate-limit 8, Zod 4, @google/genai 2
Data	SQLite via Prisma 5.22
ML service	FastAPI 0.115, scikit-learn 1.5.2, PyTorch 2.5.1, sentence-transformers 3.3.1, faiss-cpu 1.8.0
Sandbox image	Alpine with Node 20, Python 3.12, OpenJDK 21, GCC 13
Testing	pytest 8.3 (ML); node:test with tsx (API and runner)

VI. ADVANTAGES

- **Unified preparation workflow:** practice, learning, interviews, resume feedback, contests and community share one account and one progress model.
- **Actionable readiness guidance:** the score is app-relative, monotone in effort, and accompanied by ranked next steps instead of a bare number.
- **Protected test data and safe execution:** per-submission container isolation, server-side expected outputs and first-solve-only XP.
- **Graceful degradation:** with the ML service or Gemini offline, every feature falls back to a labelled deterministic path rather than failing.
- **Honest AI:** demo status is flagged in every placement response, and scorers return errors rather than fabricated values.
- **Modular and portable:** the ML service is independent of the API, and the whole stack runs from a single setup command without external databases.
- **Measured, reproducible evaluation:** every number in Section VII comes from a seeded script or an automated suite in the repository.

VII. EXPERIMENTAL EVALUATION

A. Setup

Experiments ran on a Windows 11 laptop (Docker Desktop with WSL2, Python 3.10, Node.js 24) with fixed random seeds. Model selection used cross-validation on the training split only; reported figures come from a single evaluation on the held-out split. System benchmarks ran the real Express server, the real ML service and the Docker runner against a throw-away SQLite database seeded with the full bank. TABLE XVI and Fig. 10 summarise the models.

TABLE XVI. MODEL EVALUATION SUMMARY

Model	Data and selection	Held-out result
Placement readiness (gradient boosting, monotone)	8,000 synthetic users; 6,400 train / 1,600 test; CV ROC-AUC 0.864	Accuracy 0.776; ROC-AUC 0.860; Brier 0.152
Difficulty (TF-IDF + naive Bayes)	297 curated examples; 237 train / 60 test; CV accuracy 0.760	Accuracy 0.767; macro-F1 0.767
Interview scorer	Untrained; 70 semantic + 30 keyword points	See Section VII-F
Recommender (TF-IDF cosine)	Profile = mean of solved problems	Precision@10 0.80 (topic)
RAG retrieval (FAISS)	101 chunks; 30 queries	hit@1 0.83; MRR 0.87

TABLE XVII. MODEL CONFIGURATION

Component	Settings
Placement GBM	HistGradientBoosting: 200 iterations, depth 2, learning rate 0.05, min leaf 40, monotone +1 on six features, seed 42
Placement LR	StandardScaler + LogisticRegression (2,000 iterations)
Difficulty	TF-IDF (1,500 terms, 1-3-grams, sublinear); LR C = 1; LinearSVC C = 0.5 calibrated; MNB alpha 0.1; balanced classes
Recommender	TF-IDF (5,000 terms, 1-2-grams, sublinear, English stop words); cosine to mean profile
Encoder / index	all-MiniLM-L6-v2 (384-d); FAISS IndexFlatIP on L2-normalised vectors
Generation	Gemini 2.5 Flash, temperature 0.3, 1,024 tokens, no thinking budget, 12 s limit
Protocol	Stratified 80/20 split; 5-fold CV on the training split; seed 42



Fig. 10. (a) Held-out metrics; (b) difficulty confusion matrix; (c) interview scorer probes.

B. Placement-readiness model

On the 1,600 held-out users the model reaches accuracy 0.776 (95% bootstrap CI 0.757-0.797), ROC-AUC 0.860 (CI 0.844-0.877), precision 0.775, recall 0.776 and Brier score 0.152 (TABLE XVIII). The confusion matrix is [[620, 180], [179, 621]]. The most informative reference is the *oracle*: the noise-free signal s that generated the labels achieves a test ROC-AUC of only 0.861, because the labels contain logistic noise. The model's 0.860 is therefore at the ceiling of what these labels allow, and the remaining error is label noise, not model weakness.

TABLE XVIII. PLACEMENT MODEL: GRADIENT BOOSTING VERSUS LOGISTIC REGRESSION

Metric (test)	Grad. boosting	Logistic reg.	Oracle signal
ROC-AUC	0.860	0.861	0.861
Accuracy	0.776	0.778	-
Brier score	0.152	0.151	-
Monotone in all features	yes	no	yes

Logistic regression is marginally better on the test split (AUC 0.861) but violates monotonicity in practice_volume, so the constrained model is preferred; the difference of 0.001 AUC is far inside the confidence interval. Calibration is good

(expected calibration error 0.031; Fig. 11c), so the probability can be shown as a score.

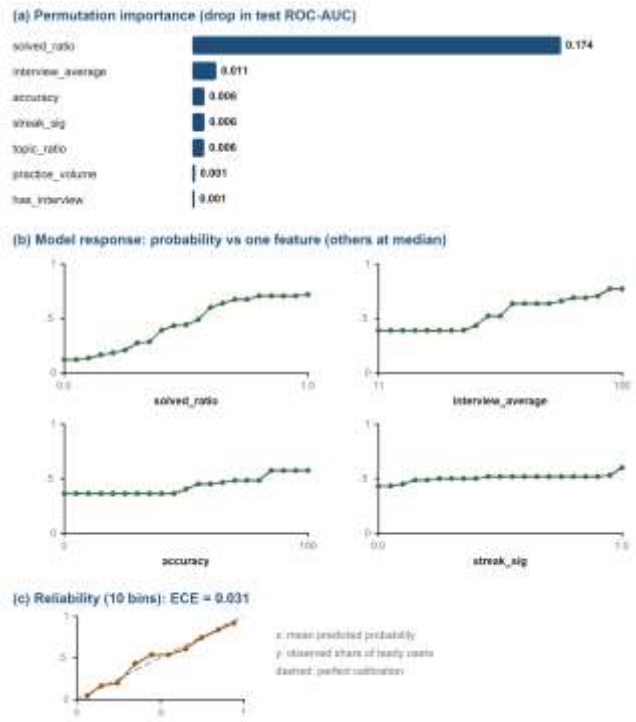


Fig. 11. Placement model: (a) permutation importance, (b) response curves, (c) reliability.

Permutation importance (Fig. 11a) shows that solved_ratio dominates (AUC drop 0.174), followed by interview_average (0.011); accuracy, streak and topic share contribute about 0.006 each. This matches the design, where the solved share carries 35% of the signal, but also shows that the model is mostly a function of one feature. The response curves (Fig. 11b) rise monotonically and saturate where the design saturates: the topic curve flattens at about 25% of the curriculum and the practice curve at 1.5 submissions per problem.

Because features are ratios, scores do not depend on bank size: the same relative profile scores 55, 55 and 55 with banks of 21, 276 and 2,000 problems (fallback: 53, 55, 54). Across the test split the model and the fallback differ by 0.028 on average (maximum 0.194, correlation 0.993), so a student sees consistent scores whether or not the ML service is up.

TABLE XIX applies the model and the fallback to six hand-built profiles on a 276-problem bank. Scores move in the expected direction, and for most profiles the model and the fallback agree within a few points. The exception is "Accurate but few" (20 problems solved with 91% accuracy, a strong interview and a 15-day streak): the model scores 56 and calls it ready, whereas the fallback, which discounts accuracy until enough submissions exist, scores 32. Such a profile is rare in the synthetic cohort, so this is an extrapolation region where the two scorers differ by 24 points, more than the 19-point maximum seen on the test split.

TABLE XIX. SAMPLE READINESS PROFILES (BANK OF 276 PROBLEMS, 589 TOPICS)

Profile	Model	Fallback	Ready	Top insight
New account	5	4	no	Solve 69 more Arena problems to reach 25% of the bank (0/276...
Beginner	6	5	no	Complete a mock interview to include interview performance in...
Accurate but few	56	32	yes	Your 91% accuracy counts fully after 69 graded submissions (2...
Many attempts, low accuracy	8	7	no	20% of your submissions are accepted - use Run on edge cases...
Steady	41	44	no	Keep your 7-day streak going - consistency credit maxes out a...
Strong	98	98	yes	Solve 21 more Arena problems to reach 80% of the bank (200/27...

Because the signal weights were chosen by design, we also tested how sensitive the readiness estimate is to them. Each of the six weights was scaled by 0.8 and 1.2 in turn (the weights were then renormalised) and the documented fallback probability was recomputed for the 1,600 test users (TABLE XX). A 20% change in any single weight moves the score by 0.5 to 1.3 points on average, and changes the ready/not-ready decision for between 0.4% and 1.4% of users. The estimate is therefore robust to moderate disagreement about the weights, with the solved share (the largest weight) the most influential.

TABLE XX. SENSITIVITY OF THE FALLBACK SCORE TO SIGNAL WEIGHTS (1,600 TEST USERS)

Weight	-20% pts	-20% flips	+20% pts	+20% flips
Solved (0.35)	1.25	1.4%	1.05	1.2%
Accuracy (0.20)	0.65	0.8%	0.61	0.5%
Interview (0.15)	0.99	1.1%	0.95	1.4%
Streak (0.10)	0.70	0.6%	0.68	0.7%
Topics (0.10)	0.59	0.9%	0.57	0.4%
Practice (0.10)	0.54	0.5%	0.52	0.8%

To check the whole pipeline end to end, we simulated a new student through the live API (Express server, Docker runner and ML service): the account solved problems by submitting the reference JavaScript solutions, took one technical mock interview with reasonable answers, and completed learning topics with a perfect quiz score (TABLE XXI). The readiness score rose from 5 to 41 over the session. Solving the first 20 problems moved it only from 5 to 9, because 20 problems is 7% of the bank; the first interview (average 53, held down by the generic reference used for two of the five technical questions) nearly doubled it (9 to 17), since the interview carries 15% of the signal and none had been taken; and reaching 60 problems raised it to 40. Ten learning topics added 2 points. The score never decreased in this scenario, as the monotone design intends, but even 60 problems, 82% accuracy, an interview and 40 topics left the student below the 50 threshold: the streak component was limited to one day because the session took place on one date. The experiment also exercised XP, levels and badges (10 badges by the end), and shows that the number the student sees responds to every activity type.

TABLE XXI. SIMULATED STUDENT JOURNEY THROUGH THE LIVE API

Stage	Solved	Acc.	Intv.	Topics	XP / lvl	Score
New account	0	-	-	0	0 / 1	5
5 problems solved	5	56%	-	0	130 / 1	8
20 problems solved	20	67%	-	0	660 / 2	9
+ 1 mock interview	20	67%	53	0	690 / 2	17
+ 10 topics	20	67%	53	10	840 / 2	19
60 problems solved	60	82%	53	10	2420 / 5	40
+ 30 more topics (40 total)	60	82%	53	40	2870 / 6	41

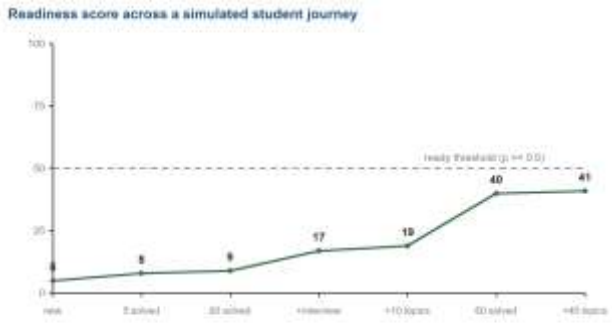


Fig. 12. Readiness score at each stage of the simulated journey.

C. Difficulty classifier

Among the three TF-IDF candidates, multinomial naive Bayes had the best cross-validated accuracy (0.760) and the best test accuracy (0.767), ahead of the calibrated linear SVM (0.750) and logistic regression (0.717). With 60 test examples the 95% Wilson interval for the winner's accuracy is 0.65-0.86, so the candidates are not statistically separable.

TABLE XXII. DIFFICULTY CLASSIFIER PER-CLASS RESULTS (TEST, N = 20 PER CLASS)

Class	Precision	Recall	F1
Easy	0.773	0.850	0.810
Medium	0.667	0.800	0.727
Hard	0.929	0.650	0.765

Hard problems have the highest precision (0.93) but the lowest recall (0.65): the model is conservative about calling a problem Hard. The most discriminative single words for Easy are "elements, check, return"; for Medium "bfs, hash, graph, tree"; and for Hard "solve, problem, dp", i.e. the classifier has learnt the vocabulary that the curated examples associate with each level. The learning curve (Fig. 13c) rises from 61.8% with 47 training examples to 73.8% with 237 and is flat beyond about 140, so more *varied* labelled data, not more of the same, is the route to improvement. The training texts are curated descriptions rather than real problem statements, so external validity is limited.

D. Problem recommender

Recommendation quality was assessed by topic coherence. For each of 17 topics with at least eight problems we sampled three solved problems 50 times, recommended ten unsolved problems and measured the share carrying the topic tag. The content-based recommender achieves a macro-average precision@10 of 0.80, against 0.05 for random selection and 0.06 for the cold-start (easiest-first) list (Fig. 13a). Topics with the best coherence are Linked Lists (1.00), Bit Manipulation (0.98), Dynamic Programming (0.97); the weakest are Two Pointers (0.52), Arrays (0.60), Recursion (0.61), whose problems share vocabulary with many other topics. A recommendation takes 0.36 ms. The metric is partly circular, because tags are part of the text that is vectorised, and measures topical coherence, not learning benefit; a user study is needed to show the latter.

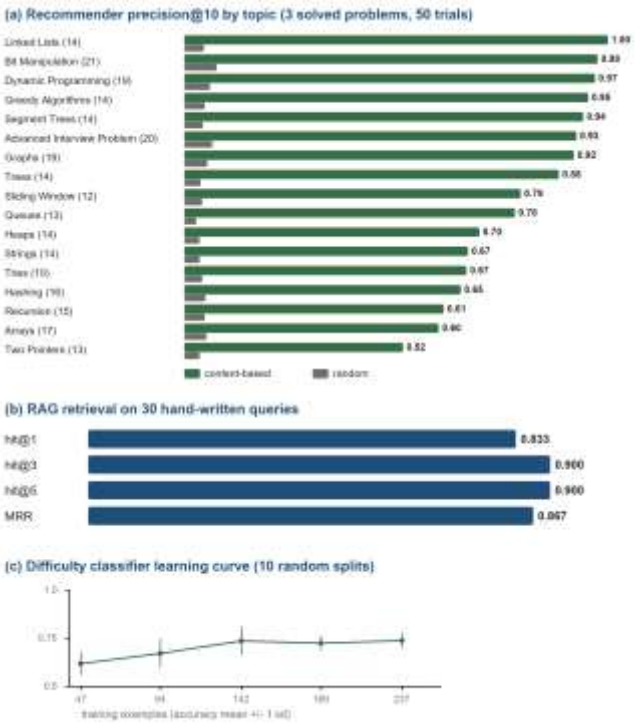


Fig. 13. (a) Recommender precision@10 by topic; (b) RAG retrieval; (c) difficulty-classifier learning curve.

The recommender is difficulty-blind. For a user who solved Two Sum, Valid Anagram and Group Anagrams, the first four recommendations are Count of Smaller Numbers After Self (Hard); Modular Exponentiation (Fast Power) (Easy); Subarray Sum Equals K (Medium); Best Time to Buy and Sell Stock (Easy). The ranking is topically plausible, but it offers a Hard segment-tree problem to a beginner, because only text similarity is used. A difficulty-aware re-ranking (for example penalising problems more than one level above the user's recent solves) is an obvious next step.

E. Retrieval for the mentor

We wrote 30 natural-language questions, each with one expected knowledge-base topic (for example "How do I detect a cycle in a linked list?" and "Linked Lists"), and retrieved the top five chunks from the 101-chunk index. The expected topic is first for 83.3% of queries, within the top three for 90.0% and within the top five for 90.0%; the mean reciprocal rank is 0.867 (Fig. 13b) and the mean top-1 cosine similarity is 0.60. Retrieval takes 12.4 ms on average (13.7 ms at the 95th percentile), and a complete local-synthesis answer 23.1 ms.

All five first-rank misses are informative. For the BFS/DFS and binary-search questions the expected chunk was second. For "find the K largest elements", "generate all permutations" and "longest substring without repeating characters" the top result was a problem-bank chunk on exactly that topic (for example Top K Frequent Elements and Longest Substring Without Repeating Characters), which in our reading is a good answer although the strict metric counts it as a miss. The evaluation set is small and was written by the authors, so it indicates feasibility, not generalisation.

F. Interview-answer scorer

We wrote six questions (three technical, one HR, two behavioural) and, for each, a strong, a partial and an off-topic answer, plus keyword-stuffed answers for three technical questions. TABLE XXIII and Fig. 14 report the results. Strong answers average 60.8, partial 40.2 and off-topic 11.0. The full score orders the three quality levels correctly in 94.4% of 18

pairwise comparisons, against 88.9% for the semantic part alone and 72.2% for the keyword part alone, so both parts help.

TABLE XXIII. INTERVIEW SCORER BY ANSWER TYPE

Answer type	n	Mean (min-max)	Semantic / keyword
Strong	6	60.8 (21-84)	42.2 / 18.6
Partial	6	40.2 (29-69)	30.2 / 10.0
Off-topic	6	11.0 (5-21)	5.8 / 5.4
Keyword-stuffed	3	50.0 (25-63)	31.3 / 18.8

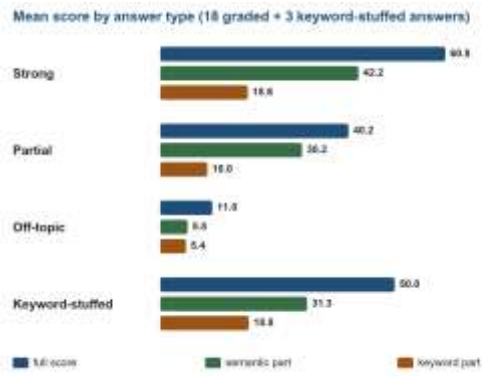


Fig. 14. Interview scorer: mean score and its two components by answer type.

Two weaknesses stand out. First, *keyword stuffing works*: lists of repeated technical terms average 50.0, above the 40.2 of honest but partial answers, because the semantic part still rewards topical vocabulary and the keyword part rewards density. Second, *reference coverage drives accuracy*. Only 5 of the 15 stock interview questions match a question-specific reference set: three of five technical, one of five HR and one of five behavioural; the rest use the generic reference, and two behavioural questions ("a project that failed", "learn a new technology") are wrongly matched to the "tell me about yourself" reference through shared words (the correct "project failed" reference is shadowed because it is tested later). Where a specific reference exists, strong answers average 76.5; where only the generic reference applies (hash-table complexity and tight deadline), the strong answers score 21 and 38. Scores are therefore trustworthy only for questions with curated references, and the UI should say so. Scoring takes 19.1 ms per answer once the encoder is loaded.

G. System performance

TABLE XXIV reports end-to-end latencies measured over loopback with the server and ML service running (n = 20 to 100 after a warm-up request). Light endpoints respond in about 15 ms (the median is quantised by the roughly 15 ms timer resolution of Windows). Login costs 109 ms by design, because PBKDF2 with 210,000 iterations is deliberately slow; at four concurrent logins the server sustains 32.7 logins per second. Dashboard analytics, which makes two ML calls, takes 47 ms, a RAG mentor answer 45 ms and a scored interview answer 81 ms. With ten concurrent clients the server handles about 1369 requests per second for a problem detail.

TABLE XXIV. ENDPOINT LATENCY (MS) AND PAYLOAD

Endpoint	p50	p95	Payload
GET /health	28	31	285 B
POST /auth/login	109	120	550 B
GET /auth/me	15	17	323 B
GET /problems/:id	15	17	3.3 KB
GET /problems (list)	339	525	968 KB
GET /learning-tracks	15	16	128 KB
GET /dashboard/analytics	47	83	4.1 KB
POST /mentor/ask (local)	45	59	1.3 KB
POST /resume/analyze	15	17	986 B
POST /interview answer	81	141	-

The slowest light endpoint is the problem list: it returns about 968 KB and takes 339 ms at the median, because it ships every problem with starter code, reference solutions and examples. Paginating the list and fetching details on demand would cut the payload by an order of magnitude. Code execution is dominated by container start-up (TABLE XXV): a Run of two visible tests takes about 587-664 ms for Python and JavaScript, 614 ms for C and 1502-1596 ms for Java and C++ (compilation). Eight simultaneous JavaScript runs finished in 1.73 s in total, showing the queue behind the four-container limit.

TABLE XXV. SANDBOX RUN LATENCY, TWO SAMPLE TESTS (MS)

Language	p50	p95	First	Verdict
JavaScript	664	689	932	Accepted
Python	587	733	582	Accepted
C	614	679	697	Accepted
Java	1502	1538	1578	Accepted
C++	1596	2047	2624	Accepted

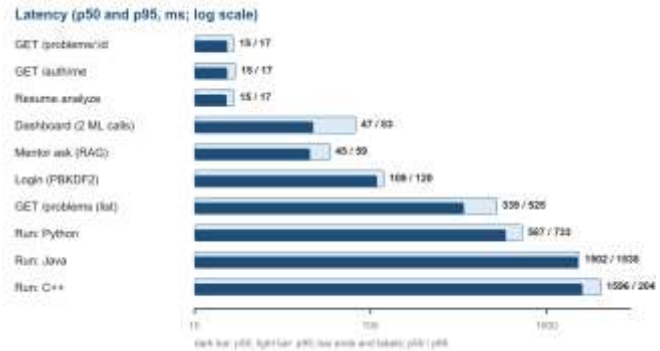


Fig. 15. Median and 95th-percentile latency of representative operations.

H. Sandbox security evaluation

The Docker-runner suite drives the runner directly and probes the container boundary with hostile programs (TABLE XXVI). All 60 checks passed in the latest run. Verdict checks ran for each of the five languages; isolation checks ran for the languages whose probes differ.

TABLE XXVI. SANDBOX ATTACK AND VERDICT BATTERY (60 CHECKS, ALL PASSED)

Probe	Expected and observed behaviour
Correct solution (per language)	Accepted, hidden tests included
Wrong answer / runtime / compile error	Correct classification; sandbox paths stripped from messages
Infinite loop	Time limit exceeded; other submissions unaffected
Network, DNS, writes outside /tmp, host secrets	All blocked
Fork bomb	Contained by PID limit and cleaned up
Memory hog	Killed at 256 MB
Output flood	Capped at 64 KB
Reading other tests, forging results, killing the driver	Impossible; no state survives between tests
Deep /tmp nesting with removed permissions	/tmp fully reset between tests
Container that outlives its deadline	Force-killed by the server
Server environment variables	Never passed to containers
After the run	No sandbox containers left behind

I. Automated testing

The ML service has 103 pytest cases covering authentication and health, inference, placement logic, RAG, training and administration; all 103 passed. The API suite (39 test cases) exercises registration and JWT forgery, role checks, problem CRUD, runner verdicts, sandbox escapes on the local runner, mock interviews, contests, discussions, resume analysis and learning tracks. Of these, 37 passed and 2 failed in the latest run. One assertion hard-codes the number of Tree problems (3), which no longer matches the expanded bank (14). The second is the audit that executes every reference solution against its stored tests: it flagged 106 of 552 problem-language checks (19%), caused by incomplete or inconsistent hidden-test data in recently added problems.

TABLE XXVII. AUTOMATED TEST SUITES

Suite	Scope	Outcome (this study)
ML service (pytest)	103 cases	103 passed
API end-to-end (node:test)	39 cases, temp database	37 passed, 2 failed
Docker runner (node:test)	60 checks, 5 languages	60 of 60 passed

TABLE XXVIII. ML SERVICE TEST INVENTORY

File	Cases	Focus
test_auth_and_health	30	API key, validation, health
test_inference	25	Endpoints and edge cases
test_placement	19	Features, monotonicity, insights
test_training	12	Training pipelines
test_rag	11	Index, retrieval, answers
test_admin	6	Retrain, rebuild

The 106 failing problem-language checks of the reference-solution audit deserve a closer look. The log lists 100 of them in full (50 problems, each failing in both JavaScript and Python); the remaining 6 (about three problems) were truncated from the log. Because the JavaScript and Python references fail in the same way, the fault lies in the stored test data and not in the solutions. TABLE XXIX classifies the 50 listed problems.

TABLE XXIX. CLASSIFICATION OF THE 50 LISTED AUDIT FAILURES

Failure type	Problems	Typical cause
Placeholder expected output	11	Test 1 expects "TODO" or "@"
Runtime error on a late hidden test	24	Malformed or mismatched test input
Wrong answer on a late hidden test	15	Wrong expected output or format
By topic (largest)	-	Advanced 8, Recursion 6, Two Pointers 6, Hashing 5

Eleven problems still carry placeholder expected outputs, which would make every correct submission fail; the other 39 fail only on one of the last hidden tests (cases 5 to 9), so students would see sample tests pass and then an unexplained hidden-test failure. Because hidden failures reveal no detail by design, the platform cannot help a student diagnose such a fault, which makes correct test data a prerequisite for the hidden-test policy.

J. Discussion and threats to validity

The measurements support three conclusions. The architecture works: components that are separate processes respond in tens of milliseconds, and the sandbox adds roughly half a second to a second per execution. The ML components do what they were designed to do on the data available: the readiness model is at the ceiling of its noisy labels, retrieval is accurate on a small query set, and the recommender is topically coherent. And the remaining weaknesses are in data, not algorithms: reference coverage for interview scoring, labelled data for difficulty, and test-data quality in the problem bank.

Several practical lessons follow. (1) Synthetic labels are useful for engineering a pipeline but their ceiling AUC is the right yardstick, not an absolute one. (2) Monotonic constraints cost almost nothing in accuracy and remove a class of embarrassing behaviour. (3) A deterministic fallback should be validated against the model, as we did, because users will see whichever is active. (4) Heuristic scorers need adversarial tests; the keyword-stuffing result would not have appeared on honest answers alone. (5) Hidden-test policies depend on data quality, so an automated audit that runs every reference solution should gate each release of the problem bank.

Threats to validity are as follows. *Construct validity*: the placement label is a synthetic rule, so the model's accuracy measures its ability to recover that rule, not to predict employment. *Internal validity*: evaluation sets for retrieval and interview scoring were written by the authors who built the systems. *External validity*: one machine, one operating system, and a seeded problem bank; the absolute latencies will differ elsewhere, and the loopback timer resolution limits the precision of the light-endpoint figures. *Statistical power*: the difficulty test set has 60 examples and the interview study 21 answers, so differences of a few points are not significant.

VIII. LIMITATIONS AND ETHICAL CONSIDERATIONS

The placement model is trained on synthetic data and has not been validated against real placement outcomes, so its absolute accuracy says nothing about real-world prediction; every response is labelled as a demonstration. The difficulty classifier is trained on fewer than 300 curated examples. The interview scorer depends on a small set of curated reference answers and can be gamed by keyword stuffing. Company tags are partly generated by a deterministic fallback and must not be read as hiring frequency. Without a Gemini key, learning-track topic pages use static text that is generic to the track rather than specific to the topic.

The problem bank needs a data-quality pass. The problem bank also needs a data-quality pass: the reference-solution audit above shows that about 19% of problem-language checks fail on the stored test data, and those entries must be corrected before wider deployment. The problem endpoints also return reference solutions and editorials to any caller, including unauthenticated ones; this supports learning but means that first-solve XP does not prove independent work. A production deployment that wants competitive integrity would gate solutions behind authentication, an unlock rule or the contest window.

Predicting a student's "readiness" carries risks. A low score can discourage, a high score can create false confidence, and any model fitted to real outcomes may encode historical bias in who was placed. The system therefore explains scores with ranked, controllable actions, avoids protected attributes, and labels the model as a demonstration; any deployment on real data should add consent, anonymisation, bias auditing and human oversight. Resume text and answers are personal data and should be handled under the institution's data-protection policy; the optional Gemini calls send that text to an external service, which users must be told.

IX. CONCLUSION

Placify AI shows that a placement-preparation platform can integrate coding practice, learning, interview rehearsal and resume feedback while adding machine-learning guidance that is transparent about its limits. Separating a FastAPI intelligence service from the Express transactional API, isolating untrusted code in per-submission containers, and giving every AI feature a deterministic fallback make the system safe to run and easy to evolve.

On held-out data the readiness model reached 0.860 ROC-AUC (at the ceiling of its noisy labels) and the difficulty classifier 76.7% accuracy; retrieval placed the expected chunk first for 83.3% of queries; the ML service passed all 103 of its tests and the sandbox all 60 of its checks. The evaluation also exposed concrete weaknesses: keyword stuffing, limited reference coverage for interview scoring, an oversized problem-list payload, and inconsistent test data in part of the problem bank. The readiness score is trained on synthetic data and must be re-fitted and validated on real placement outcomes before it is used to advise students or institutions.

X. FUTURE WORK

The most valuable next step is collecting consented, anonymised placement outcomes to replace the synthetic cohort, calibrating the readiness model and measuring whether its insights change student behaviour in a controlled study. The interview scorer should gain a larger, rubric-based reference bank, a cross-encoder or language-model judge for relevance, and a keyword-stuffing penalty; the difficulty classifier needs more and more varied labelled problems or a fine-tuned embedding model.

Engineering work includes correcting the inconsistent problem-bank test data and updating the stale API assertion; paginating the problem list and gating reference solutions; a PostgreSQL option for multi-instance deployment; plagiarism detection and live scoreboards for contests; personalised learning paths driven by the recommender; speech-based mock interviews; mobile clients; and stronger runner isolation with gVisor. Observability (metrics, centralised logging and tracing) and penetration testing should precede any large-scale rollout.

APPENDIX A. INTERVIEW QUESTIONS AND REFERENCE MATCHING

TABLE XXX lists the 15 stock questions and the reference set the scorer selects for each. "generic" is the default or behavioural reference; "spurious" marks a question matched to the wrong specific reference because the key "tell me about yourself" is tested first and shares the words "tell", "me" and "about".

TABLE XXX. STOCK QUESTIONS AND MATCHED REFERENCE

Type	Question	Reference
Tech	SQL versus NoSQL; when to choose each	specific
Tech	Hash-table search complexity and collisions	generic
Tech	Process versus thread; OS scheduling	specific
Tech	Design a rate limiter for a public REST API	specific
Tech	What happens when you type a URL and press Enter	generic
HR	Tell me about yourself	specific
HR	Why do you want to join our company	generic
HR	Strengths and one weakness	generic
HR	Where do you see yourself in five years	generic
HR	Relocation, shifts or service agreement	generic
Behav.	Conflict with a teammate	specific
Behav.	A project that failed; what you learnt	spurious
Behav.	Taking initiative without being asked	generic
Behav.	Meeting a tight deadline	generic
Behav.	Learning a new technology quickly	spurious

APPENDIX B. REPRODUCIBILITY

All results can be regenerated from the repository. The commands below assume Windows with Node.js 20+, Python 3.10-3.12 and, for the sandbox, Docker with the runner image.

- npm install and npm run setup create the environment file, apply the schema and seed the bank.
- ml_service/start.bat creates the virtual environment, installs pinned requirements and trains missing models; models/train_all.py retrains every artifact with fixed seeds.
- npm run runner:build builds the sandbox image; npm run test:runner runs the 60 sandbox checks.
- npm test runs the API suite against a temporary database; python -m pytest in ml_service runs the 103 ML tests.
- The evaluation scripts used for Section VII (permutation importance, calibration, retrieval, scorer ablation, latency and screenshot capture) are small stand-alone programs that import the same modules as the service.

APPENDIX C. EXAMPLE ML SERVICE EXCHANGES

The listings below are verbatim outputs of the trained models (long strings shortened). The mentor example was produced with a Gemini key configured, so its method is "RAG + Gemini"; without a key the same request returns the extractive local synthesis.

Listing 1. Placement score request and response

```
POST /ml/placement-score
{ "problems_solved": 60, "total_problems": 276,
  "submission_count": 140, "accuracy": 60, "streak": 7,
  "interview_average": 65, "topics_completed": 60,
  "total_topics": 589 }
->
{ "placement_ready": false, "score": 41,
  "probability": 0.4085, "model": "HistGradientBoosting",
  "insights": [
    "[Consistency] Keep your 7-day streak going - consistency...",
    "[Learning] Finish 88 more learning-track topics to reach...",
    "[Accuracy] 60% of your submissions are accepted - use Run...",
    "[Practice] Solve 9 more Arena problems to reach 25% of th..."
  ] }
```

Listing 2. Difficulty prediction

```
POST /ml/difficulty-predict (Two Sum; Arrays, Hashing)
-> { "difficulty": "Easy", "confidence": 0.9693,
     "probabilities": { "Easy": 0.9693,
       "Medium": 0.0290, "Hard": 0.0017 } }
POST /ml/difficulty-predict (Median of Two Sorted Arrays)
-> { "difficulty": "Medium", "confidence": 0.5839,
     "probabilities": { "Easy": 0.0508,
       "Medium": 0.5839, "Hard": 0.3653 } }
```

Listing 3. Interview answer score

```
POST /ml/interview-score (SQL vs NoSQL; 36-word answer)
-> { "score": 84, "semantic_similarity": 0.9021,
     "feedback": "Excellent response! Strong depth, well-...",
     "method": "all-MiniLM-L6-v2" }
```

Listing 4. Mentor answer

```
POST /rag/mentor-ask (cycle in a linked list?)
-> { "method": "RAG + Gemini",
     "answer": "Detecting a cycle in a linked list is a common probl
       em, often solved using the Floyd's Tortoise and Hare
algorit...",
     "sources": [ "DSA - Linked Lists (0.55)", ... ] }
```

APPENDIX D. RETRIEVAL EVALUATION QUERIES

TABLE XXXI lists the 30 evaluation queries with the rank of the first chunk whose topic equals the expected topic (">5" means it was not among the top five).

TABLE XXXI. RETRIEVAL QUERIES AND RANK OF THE EXPECTED TOPIC

Query	Expected topic	Rank
How do I detect a cycle in a linked list?	Linked Lists	1
What is the difference between BFS and DFS and when do I us...	Graphs: BFS, DFS, Shortes...	2
Explain memoization versus tabulation	Dynamic Programming	1
How does a hash map deal with collisions?	Hashing & Hash Maps	1
What is a monotonic stack used for?	Stacks & Queues	1
How can I find the K largest elements efficiently?	Heaps & Priority Queues	>5
How do I generate all permutations of a list?	Backtracking	>5
What is the time complexity of binary search?	Binary Search	2
Which data structure supports fast prefix autocomplete?	Tries (Prefix Trees)	1
How can I answer range sum queries with point updates?	Segment Trees & Fenwick T...	1
How do I check whether a number is a power of two using bit...	Bit Manipulation	1
Which sorting algorithms run in n log n?	Sorting Algorithms	1
What are decorators in Python?	Decorators & Closures	1
How do generators save memory?	Generators & Yield Statem...	1
How do async and await work in Python?	Asyncio & Async Programmi...	1
What do *args and **kwargs mean?	Functions & Variable Argu...	1
How would I use Redis to cache database queries?	Caching & Redis	1
How does a load balancer distribute requests?	Load Balancers & Scaling	1
How should I shard a large database?	Database Design & Sharding	1
What are good REST API design practices?	API Design & REST	1
How do message queues enable event driven systems?	Message Queues & Event-Dr...	1
How should I prepare for the TCS Ninja test?	TCS Ninja/Digital Prepara...	1
How do I prepare for an Amazon SDE interview?	Amazon SDE Interview Prep...	1
How should I structure my resume so an ATS can parse it?	Resume & ATS Optimization	1
How do I answer the tell me about yourself question in an H...	HR Interview Preparation	1
Which algorithm searches for a pattern in a string in linea...	Strings & Pattern Matching	1
How do I find the longest substring without repeating chara...	Arrays & Sliding Window	>5
Why does an in-order traversal of a BST give sorted output?	Binary Trees & BST	1
How does method resolution order work with multiple inherit...	OOP: Classes, Inheritance...	1
How is scikit-learn cross validation done?	Machine Learning: Regress...	1

APPENDIX E. INTERVIEW SCORER ANSWERS

TABLE XXXII gives the score of each of the 21 answers used in Section VII-F. Semantic similarity is the best cosine to the matched references and keyword is the clamped keyword-and-length score of Eq. (5).

TABLE XXXII. SCORES OF THE 21 EVALUATION ANSWERS

Question	Answer type	Score	Semantic	Keyword
SQL vs NoSQL	Strong	84	0.85	0.80
SQL vs NoSQL	Partial	69	0.72	0.61
SQL vs NoSQL	Off-topic	8	0.03	0.18
SQL vs NoSQL	Keyword-stuffed	62	0.61	0.66
Hash table	Strong	21	0.06	0.57
Hash table	Partial	33	0.20	0.62
Hash table	Off-topic	11	0.09	0.18
Hash table	Keyword-stuffed	25	0.06	0.69
Rate limiter	Strong	76	0.78	0.70
Rate limiter	Partial	48	0.64	0.11
Rate limiter	Off-topic	5	0.00	0.18
Rate limiter	Keyword-stuffed	63	0.67	0.53
About yourself	Strong	72	0.73	0.69
About yourself	Partial	32	0.43	0.08
About yourself	Off-topic	21	0.23	0.18
Conflict	Strong	74	0.81	0.57
Conflict	Partial	29	0.38	0.08
Conflict	Off-topic	9	0.05	0.18
Deadline	Strong	38	0.38	0.39
Deadline	Partial	30	0.22	0.50
Deadline	Off-topic	12	0.10	0.18

APPENDIX F. ABBREVIATIONS

TABLE XXXIII. ABBREVIATIONS USED IN THIS PAPER

Term	Meaning
API	Application programming interface
ATS	Applicant tracking system
AUC / ROC	Area under the receiver operating characteristic curve
CV	Cross-validation
ECE	Expected calibration error
FAISS	Facebook AI Similarity Search library
GBM	Gradient boosting machine
JWT	JSON Web Token
MRR	Mean reciprocal rank
PBKDF2	Password-based key derivation function 2
PID	Process identifier
RAG	Retrieval-augmented generation
SPA	Single-page application
TF-IDF	Term frequency-inverse document frequency
XP	Experience points (gamification)
ORM	Object-relational mapper
CRUD	Create, read, update, delete

APPENDIX G. PROBLEMS FLAGGED BY THE REFERENCE-SOLUTION AUDIT

TABLE XXXIV lists the 50 problems whose failures appear in full in the audit log, with the failure type and the number of the first failing test. Problems with a placeholder expected output are listed first. About three further problems (6 of the 106 failing checks) were truncated from the log and are not listed; re-running the audit after the fixes will enumerate any that remain.

TABLE XXXIV. PROBLEMS FLAGGED BY THE AUDIT (JAVASCRIPT AND PYTHON REFERENCES BOTH FAIL)

Problem id (prob-...)	Failure	Test
advanced-capacity-to-ship-packages-within-d-days	placeholder expected output	1
advanced-design-hit-counter	placeholder expected output	1
advanced-find-peak-element	placeholder expected output	1
advanced-insert-delete-getrandom-o-1	placeholder expected output	1
advanced-search-in-rotated-sorted-array	placeholder expected output	1
advanced-set-matrix-zeroes	placeholder expected output	1
recursion-binary-strings-without-adjacent-ones	placeholder expected output	1
recursion-generate-all-subsequences-of-a-string	placeholder expected output	1

Problem id (prob-...)	Failure	Test
recursion-gray-code	placeholder expected output	1
recursion-josephus-problem	placeholder expected output	1
recursion-pascal-s-triangle	placeholder expected output	1
advanced-logger-rate-limiter	runtime error	7
advanced-rotate-image	runtime error	8
arrays-merge-intervals	runtime error	9
arrays-next-permutation	runtime error	9
arrays-product-of-array-except-self	runtime error	8
arrays-rotate-array	runtime error	9
bitManipulation-counting-bits	runtime error	8
greedy-activity-selection	wrong answer	8
hashing-count-pairs-with-given-sum	wrong answer	8
hashing-first-repeating-element	wrong answer	8
hashing-insert-delete-getrandom-o-1	runtime error	9
hashing-logger-rate-limiter	runtime error	7
hashing-most-common-word	wrong answer	8
heaps-merge-k-sorted-lists	runtime error	9
linkedLists-add-one-to-a-number-stored-in-a-linked-list	runtime error	9
linkedLists-middle-of-the-linked-list	runtime error	8
linkedLists-remove-nth-node-from-end-of-list	runtime error	8
linkedLists-reorder-list	runtime error	9
queues-moving-average-from-data-stream	runtime error	8
recursion-reverse-a-string-using-recursion	runtime error	8
segmentTrees-matrix-block-sum	wrong answer	8
segmentTrees-product-of-the-last-k-numbers	wrong answer	7
segmentTrees-range-addition	wrong answer	7
segmentTrees-range-sum-query-2d-immutable	runtime error	5
slidingWindow-maximum-average-subarray-i	wrong answer	9
slidingWindow-maximum-number-of-vowels-in-a-substring-of-given-length	wrong answer	9
slidingWindow-subarray-with-given-sum	wrong answer	6
stacks-daily-temperatures	runtime error	9
strings-add-binary	runtime error	8
strings-reverse-words-in-a-string	runtime error	9
strings-valid-palindrome-ii	wrong answer	9
trees-binary-tree-right-side-view	runtime error	8
tries-index-pairs-of-a-string	runtime error	7
twoPointers-backspace-string-compare	wrong answer	9
twoPointers-merge-sorted-array	wrong answer	7
twoPointers-move-zeroes	runtime error	9
twoPointers-remove-duplicates-from-sorted-array	wrong answer	7
twoPointers-remove-element	wrong answer	7
twoPointers-segregate-even-and-odd-numbers	runtime error	7

REFERENCES

[1] React Documentation, "React," Meta Open Source. Available: https://react.dev/

[2] Express Documentation, "Express - Node.js web application framework," OpenJS Foundation. Available: https://expressjs.com/

[3] Prisma Documentation, "Prisma ORM," Prisma Data, Inc. Available: https://www.prisma.io/docs

[4] FastAPI Documentation, "FastAPI - Modern Web Framework for Building APIs with Python." Available: https://fastapi.tiangolo.com/

[5] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," J. Mach. Learn. Res., vol. 12, pp. 2825-2830, 2011.

[6] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," Ann. Statist., vol. 29, no. 5, pp. 1189-1232, 2001.

[7] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Inf. Process. Manage., vol. 24, no. 5, pp. 513-523, 1988.

[8] A. McCallum and K. Nigam, "A comparison of event models for naive Bayes text classification," in AAAI-98 Workshop on Learning for Text Categorization, 1998.

[9] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in Proc. EMNLP-IJCNLP, 2019, pp. 3982-3992.

[10] W. Wang et al., "MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," in Proc. NeurIPS, vol. 33, 2020.

[11] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with GPUs," IEEE Trans. Big Data, vol. 7, no. 3, pp. 535-547, 2021.

[12] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Proc. NeurIPS, vol. 33, 2020, pp. 9459-9474.

[13] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, Internet Engineering Task Force, 2015.

[14] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Internet Engineering Task Force, 2017.

[15] OWASP Foundation, "OWASP API Security Top 10." Available: https://owasp.org/www-project-api-security/

[16] Docker Documentation, "Docker Engine security." Available: https://docs.docker.com/engine/security/

[17] Judge0, "Judge0 - open-source online code execution system." Available: https://github.com/judge0/judge0

[18] Google, "Gemini API documentation." Available: https://ai.google.dev/gemini-api/docs

[19] J. Lewis and M. Fowler, "Microservices," martinfowler.com, 2014. Available: https://martinfowler.com/articles/microservices.html

[20] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," ACM Comput. Surv., vol. 51, no. 1, art. 3, 2018.

[21] S. Kurnia, R. Lim, and B. Cheang, "Online judge," Comput. Educ., vol. 36, no. 4, pp. 299-310, 2001.

[22] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux J., vol. 2014, no. 239, 2014.

[23] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining gamification," in Proc. 15th Int. MindTrek Conf., 2011, pp. 9-15.

[24] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work? A literature review of empirical studies on gamification," in Proc. 47th Hawaii Int. Conf. Syst. Sci. (HICSS), 2014, pp. 3025-3034.

[25] C. Romero and S. Ventura, "Educational data mining: A review of the state of the art," IEEE Trans. Syst., Man, Cybern. C, vol. 40, no. 6, pp. 601-618, 2010.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, 2019, pp. 4171-4186.

[27] V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," in Proc. EMNLP, 2020, pp. 6769-6781.

[28] Z. Ji et al., "Survey of hallucination in natural language generation," ACM Comput. Surv., vol. 55, no. 12, art. 248, 2023.

[29] G. W. Brier, "Verification of forecasts expressed in terms of probability," Mon. Weather Rev., vol. 78, no. 1, pp. 1-3, 1950.

[30] A. Niculescu-Mizil and R. Caruana, "Predicting good probabilities with supervised learning," in Proc. 22nd Int. Conf. Mach. Learn. (ICML), 2005, pp. 625-632.

[31] L. Breiman, "Random forests," Mach. Learn., vol. 45, no. 1, pp. 5-32, 2001.

[32] M. Gupta et al., "Monotonic calibrated interpolated look-up tables," J. Mach. Learn. Res., vol. 17, no. 109, pp. 1-47, 2016.

[33] Gemini Team, Google, "Gemini: A family of highly capable multimodal models," arXiv:2312.11805, 2023.

[34] SQLite Documentation, "SQLite." Available: https://www.sqlite.org/docs.html

[35] P. A. Grassi et al., "Digital identity guidelines: Authentication and lifecycle management," NIST Special Publication 800-63B, 2017.