Problem Statement:

Many coding platforms (LeetCode, GFG, CodeChef) provide **final solutions** directly, which discourages critical thinking. Students often copy solutions without learning the actual problem solving approach.

Our Solution:

Our system will act as a **step by step coding coach**, providing students with guided hints, questions, and explanations **instead of giving full answers immediately**.

Objectives:

1. Develop a **Chrome Extension and Web Application** that seamlessly integrates with coding platforms.
2. Provide comprehensive **guidance** through AI-driven step-by-step solutions, eliminating the need for onetime fixes.
3. Monitor user progress and **construct a personalized skill profile**.
4. Recommend appropriate problems and learning paths based on identified weaknesses.
5. Ensure scalability through the utilization of cloud-based infrastructure**, DevOps practices, and AI microservices**.

Tech Stack:

- **Frontend:** React.js, Chrome Extension (Manifest V3), TailwindCSS
- **Backend:** Spring Boot (REST APIs, WebSockets, Authentication)
- **AI Service:** Python (FastAPI/Flask), Hugging Face/OpenAI APIs, Machine Learning recommendation models
- **Database:** PostgreSQL (structured data), MongoDB (logs, hints)
- **Cloud & DevOps:** Docker, Kubernetes, AWS/GCP, GitHub Actions (Continuous Integration/Continuous Deployment), Monitoring (Prometheus/Grafana)

3 Semester Execution Plan

Semester 1: Foundation & MVP (Core System)

Goal: Establish a functional base system (no AI yet, rulebased hints).

- **Deliverables:**
    - Chrome Extension: Overlay UI on coding platforms (LeetCode, GFG).

- Web Dashboard (React): User login, progress tracking, problem history.
- Backend (Spring Boot): APIs for authentication, storing user attempts, and progress.
- Database Schema (Users, Problems, Progress, Hints): Define the structure of the database.
- Initial Hinting System (Manual/RuleBased Hints): Store hints in the database.
- Cloud Deployment (Basic): Deploy backend and database on AWS/GCP.
- CI/CD Pipeline for Frontend and Backend: Implement continuous integration and continuous deployment pipelines.

Semester 2: Intelligence and AI Integration

Goal: Enhance the system with real AI capabilities and personalized learning features.

- **Deliverables:**
  - **AI Microservice (Python FastAPI):** A microservice connected to the backend.
  - **LLM Integration (Hugging Face / OpenAI):** Integration of LLMs for generating **contextual hints**.
  - **"Socratic Guidance" Module:** A module where AI asks questions instead of providing direct answers.
  - **Skill Graph Model:** A model to track strengths and weaknesses (e.g., DP weak, Arrays strong).
  - **Recommendation Engine:** An engine that suggests next problems based on the skill graph.
  - **Realtime Hinting via WebSockets:** Realtime hinting capabilities.
  - **Monitoring and Logging System:** A system for monitoring and logging user activities.

Semester 3: Scaling, Cloud, and Polish

Goal: Make the system robust, productionready, and userfriendly.

- **Deliverables:**
  - **Advanced Hint Personalization:** Finetuned models, hybrid rule systems, and AI systems for personalized hints.
  - **CloudNative Deployment:** Deployment using Docker, Kubernetes, and load balancing.
  - **Optimized AI Service:** Optimized AI service with caching, reduced latency, and model selection.
  - **ElasticSearch Integration:** Integration of ElasticSearch for fast problem and hint search.
  - **Analytics Dashboard:** A dashboard to track student improvement over time.
  - **Security Enhancements:** Security enhancements such as Spring Security and OAuth2 (if needed).
  - **Final Testing with Users:** Final testing with users (classmates, juniors) and a feedback loop.
  - **Documentation and Research Paperstyle Report:** Documentation and a research paperstyle report for Major submission.

Final Deliverables

- **Chrome Extension + Web App** (React)
- **Spring Boot Backend** with APIs
- **AI Microservice** (Python + Hugging Face/OpenAI)

- **Databases** (PostgreSQL + MongoDB)
- **Cloud Deployment** (AWS/Google Cloud Platform, containerized with CI/CD)
- **Project Report + Presentation + Live Demo**

Why This Project Is Strong

- **Novelty:** Unlike coding platforms that provide prewritten solutions, this *project emphasizes critical thinking*.
- **Realworld Relevance:** Students often resort to copying solutions. Your system aims to develop problem solving skills.
- **Scalability:** The project has the potential to evolve into a viable product, such as an educational technology startup.
- **Technical Depth: It covers a comprehensive range of topics**, including **frontend development, backend programming, artificial intelligence and machine learning, cloud computing, and DevOps** practices, which are essential components of a major project.

Tech Stack Summary

- **Frontend:** React.js, Chrome Extension, TailwindCSS
- **Backend:** Spring Boot (REST/GraphQL, Authentication, Database Integration)
- **AI/ML Service:** Python (FastAPI/Flask), Hugging Face/OpenAI, Custom Recommendation ML
- **Databases:** PostgreSQL, MongoDB
- **Cloud/DevOps:** Docker, Kubernetes, AWS/GCP, CI/CD Pipelines
- **Extras:** WebSockets for Realtime Hints, ElasticSearch for Search

Frontend (User Interface)

Purpose: Dashboard + Chrome Extension UI for hints, progress, and interaction.

- **React.js:** Main web dashboard
- **Chrome Extension (React + Manifest V3):** Popup hints and injected UI on coding sites (LeetCode, GFG, etc.)
- **Redux Toolkit / React Query:** State management and asynchronous calls
- **TailwindCSS / Material UI:** Fast and clean UI components

Backend (APIs & Business Logic)

Purpose: User authentication, progress tracking, managing hint sessions, and communication with AI service.

- **Spring Boot (Java):** Robust API layer
- **Spring Security + JWT:** Secure authentication and user management
- **REST APIs / GraphQL:** For communication with frontend and extension
- **WebSockets: Real**time hint updates or interactive sessions

AI/ML Layer (Hint Generation + Personalization)

Purpose: The “intelligence” will guide users step by step.

- **Python (FastAPI / Flask microservice):** Separate AI service (easier than embedding within Spring Boot).
- **LLM Integration:**
    - **Use OpenAI API / Hugging Face models for generating guided hints.**
    - **Finetune or create prompt engineering logic for "step-by-step guidance."**
- **Recommendation Engine:** Suggest next problems, difficulty level (could use collaborative filtering / skill graph model).

Knowledge Tracking: Build a user knowledge profile (strength/weakness detection via ML).

Deployment and Scaling:

- **Cloud Provider:** AWS / GCP / Azure (any, depending on credits or preference).
- **Containers:** Docker + Kubernetes (for backend + AI service + frontend).
- **CI/CD:** GitHub Actions / Jenkins → automated builds, tests, deployments.
- **Cloud Databases:**
    - **PostgreSQL:** Relational data (users, progress, sessions).
    - **MongoDB:** Flexible storage (problem attempts, hint logs).
- **Cloud Storage (S3/Blob):** Store logs, session histories.
- **Monitoring:** Prometheus + Grafana or CloudWatch.

Database Design (HighLevel)

- **Users:**
- id
- name
- email
- Auth_info

- **Problems:**
- id
- source (e.g., LeetCode, GFG)
- difficulty
- Tags

- **UserProgress:**
- user_id
- problem_id
- attempts
- status
- Hints_used

- **Hints:**
- problem_id
- step_number
- hint_text

- AI_source

- **SkillGraph:**
- user_id
- topic
- score

1) Capture: how to get the user's typing safely and efficiently

Goal: get enough of the student's code to analyze without being invasive or causing latency.

How (practical):

- **Chrome extension injection**: inject a content script that hooks the platform's editor (LeetCode/GFG) DOM or their editor API. Many sites use CodeMirror or Monaco — you can attach to those events. Your PDF already plans an extension and injected UI.

- **Granularity / transport**:

    - Track *deltas* (changed lines or small chunks) rather than whole files on every keystroke.

    - **Debounce** (e.g., 300–800ms) for sending updates. For immediate syntax feedback you can run local parsing (see next).

    - Use a **WebSocket** session for each active hinting session (fast bi-directional updates). (Your plan mentions realtime hinting via WebSockets).

- **Local vs remote processing**:

    - Do as much **static/syntax analysis in-browser** (low-latency): run linters or parsers compiled to WebAssembly (tree-sitter, ESLint wasm builds, etc).

    - Reserve **server/AI calls** for deeper semantic checks or LLM-generated hints.

Practical caveat: don't capture long-term data unless user consents. Send *only* the minimal snippet needed for hinting.

2) Detecting "wrong code" — what to analyze

You must detect *types* of mistakes, not just that something is wrong.

Layers of detection (fast → deep):

1. **Syntax / parse errors** (instant): use language parser/AST (tree-sitter, acorn/Esprima for JS, Python ast, JavaParser). Cheap and deterministic — do in-browser.

2. **Type / lint checks** (very fast): ESLint, mypy, Pyright, clang-tidy, javac diagnostics. Some run in-browser (pyodide/myPy wasm) or on server.

3. **Static semantic checks / anti-patterns**: detect wrong algorithmic approach markers (e.g., using nested loops where hash/memo needed) via AST pattern matching or heuristics.

4. **Behavioral / dynamic checks**: run the code on sample inputs in a **sandboxed worker** (container) to catch runtime errors, incorrect outputs or performance bottlenecks (TLE). This is needed to detect logic errors that static analysis misses.

5. **Meta-patterns from history**: compare against user's previous attempts and common error clusters to label "typical off-by-one" vs "concept gap".

Tools per language (practical choices):

- JS/TS: tree-sitter + ESLint + node sandbox.
- Python: ast + pyflakes/mypy + run inside container using timeouts.
- Java/C++: compile (javac/gcc/clang) in sandbox to collect diagnostics + clang-tidy.
- Multi-language parsers: tree-sitter supports many languages and is fast.

3) Hint generation strategies (how to turn an error into a helpful hint)

Three complementary approaches — use all three in a pipeline:

A. **Rule-based / template mapping (MVP)** — deterministic, fast.

- Map specific compiler errors or AST patterns to short hints. Example: `IndexError` → "Check your loop bounds; an index went negative or overshot the list length."
- Store curated multi-step hints per problem in DB (your Semester 1 plan).

B. **Program-synthesis / symbolic checks** — semi-automated suggestions.

- If an input-output pair fails, compute a counterexample and produce a hint like "On input X your function returns Y but expected Z; what happens near the loop where you update variable V?"
- Use lightweight test-case minimizing techniques to produce succinct failing

examples.

C. **LLM / Socratic hints (Semester 2 AI)** — flexible, conversational, deeper guidance.

- Use an LLM to *rephrase* the rule-based/diagnostic output into a student-friendly question (Socratic prompting). Example: instead of saying “You forgot to sort,” ask “If the order of elements changed, would your approach still find the correct pair? What invariant do you assume about the input?”

- **Important:** Implement **guardrails and prompting** to avoid LLMs outputting full solutions. Use progressive hinting and policy-based blocking (no code that solves the entire problem until the final unlocked hint).

Progressive hint policy (practical, pedagogically-sound):

1. **Nudge** — point at the concept (one-liner).
2. **Guided question** — ask a targeted question (Socratic).
3. **Pseudo-code hint** — show high-level steps or algorithm idea (no code).
4. **Micro-solution** — small code fragment or edge-case explanation.
5. **Full solution** — only after many hints or explicit request (or as instructor override).

4) Realtime architecture — components & data flow

High-level per your project plan (extension + backend + AI microservice + DB).

Client (Chrome extension / injected UI)

- Capture deltas, run quick local parsers (tree-sitter wasm), show inline lint marks.

- Open a WebSocket or make batched HTTP calls to backend for deeper checks/LLM hints.

Backend (Spring Boot API + session manager)

- Session manager keeps state (attempt id, hints used, last AST).

- WebSocket router forwards requests to AI microservice or to the async job runner

that executes code in a sandbox.

- Cache results for identical code snippets to avoid repeated LLM calls.

AI microservice (Python FastAPI)

- Receives sanitized problem context and diagnostics, composes prompt, returns hint(s).

- Use *prompt templates* and a *hint policy engine* that decides hint tier and redaction. (Your doc suggests this split: Spring Boot for API + Python AI microservice).

Sandboxed execution

- Use pre-warmed container workers (Docker with seccomp/gVisor/Firecracker) to run tests. Always time-limit and resource-limit. Use snapshotting to avoid cold start latency.

Data stores

- PostgreSQL for users, progress, problems, hint metadata.

- MongoDB or an append-only log for attempt histories and snapshots (as your doc suggests).

Performance optimizations

- **Local analysis** for 90% of "instant" hints.

- **Caching** of hints by (problem, diagnostic signature) to return known-good hints quickly.

- **Asynchronous LLM calls** with optimistic placeholders: show a gentle “thinking” message while server returns deep hint.

5) Tracking progress & the skill graph (what to store and how to infer weakness)

What to record (minimum):

- user_id, problem_id, timestamps
- attempts count, time per attempt, hint tiers used, which hints accepted
- failure types (syntax / runtime / wrong-output / performance)
- concept tags for the problem & per-error (DP, graph, sorting, off-by-one, pointer use)

Skill graph model (practical):

- Represent user knowledge as a vector over topics (arrays, dp, graphs, greedy). Update via Bayesian or exponential moving average: success + no hints = higher; repeated failures in same topic = lower.
- Use collaborative filtering + heuristics to recommend next problems.

Analytics & signals:

- Learning velocity (improvement per attempt), hint-dependence score (how often user needs level-3+ hints), time-to-first-correct-solution.

Your project doc references a Skill Graph and Recommendation Engine for suggesting problems — exactly where this fits in Semester 2.

6) UX & pedagogical design — how hints should feel

Design principles:

- **Scaffolded help**: progressively more revealing hints (already described).
- **Socratic style**: ask questions more than give code. Implemented as a “Socratic guidance” module in your plan.
- **Non-blocking UI**: allow users to hide hints, pin them, request next hint, or ask for explanation after submitting solution.
- **Visual cues**: inline squiggles, a hint panel with hint history, “next hint” button, and an indication of how many hints were used.
- **Confidence & undo**: let students mark a hint as “helpful” or “unhelpful” to improve the system.

7) Security, safety, privacy, and academic integrity

Execution safety:

- **Sandbox** every run. Use aggressive timeouts and resource limits (CPU, memory). Isolate network. Consider Firecracker / gVisor / sandboxed VMs for better isolation.
  Data privacy:

- Minimize what you send to the cloud. Hash/anonymize Anything PII; provide an opt-in consent screen.
  Academic integrity:

- Provide quoting/attribution: if a hint uses LLM outputs trained on public code, ensure your policy prevents direct copy of full solutions until allowed. Keep logs so instructors can see hint histories for audit.

8) Implementation roadmap — MVP → AI → scale (mapped to 3 semesters)

Your PDF already laid out a three-semester plan — this is a practical mapping I recommend:

Semester 1 (MVP — rule-based & extension):

- Chrome extension overlay + simple injected UI.
- Backend with REST + simple DB tables (Users, Problems, UserProgress, Hints).
- Rule-based hint engine (error→hint templates).
- Local parsing (tree-sitter) for instant feedback.
- Basic logging and dashboard.

Semester 2 (AI & personalization):

- Add AI microservice (FastAPI), integrate LLMs for Socratic hints.
- Skill graph, recommendation engine.
- Realtime via WebSockets for interactive sessions.

- Add sandboxed execution for dynamic checks.

Semester 3 (Scale & polish):

- Containerize, Kubernetes, caching, load balancing.
- Optimize latency, fine-tune models, advanced personalization and analytics dashboard.

9) Concrete tech choices & small implementation notes

- **Editor parsing in-browser**: tree-sitter (WASM) — supports many languages.
- **Lightweight in-browser linters**: ESLint (for JS) wasm builds, pyright in wasm for Python type checks.
- **Server-side compilers**: use Docker worker pool with pre-warmed images for each language (ubuntu + toolchain). Use results to generate dynamic hints.
- **AI microservice**: Python FastAPI, keep prompt templates in repo, use local or API LLMs (OpenAI/HF); always perform post-processing to redact full solutions unless authorized.
- **Realtime comms**: WebSockets (Spring Boot + STOMP or raw sockets) or use a Node gateway if you prefer. Your doc lists WebSockets in backend.
- **DB**: PostgreSQL for relational, MongoDB for attempt logs (as in your plan).

10) Metrics & evaluation (how you'll prove it worked)

Track and report:

- % reduction in time-to-first-correct-solution.
- Learning gain on pre/post tests (control vs hint group).
- Hint-dependence (are students improving without hints over time?).

- Student satisfaction / perceived usefulness.
- False-positive hint rate (hints that were irrelevant).

11) Risks & mitigations

- **LLM hallucinations** → Mitigate: guardrails, filter outputs, hybrid rule+LLM approach.
- **Latency hurts UX** → Mitigate: local analysis + caching + progressive UI placeholders.
- **Security from remote code execution** → Mitigate: strict sandboxing, no network, short timeouts.
- **Giving away full solutions** → Mitigate: tiered hints and policy engine preventing full-solution generation until explicit unlock.

12) Minimal example data model (short)

Users(user_id, name, email)
Problems(problem_id, source, tags, difficulty)
Attempts(attempt_id, user_id, problem_id, code_snapshot, timestamp, outcome, hints_used)
Hints(hint_id, problem_id, step_no, hint_text, hint_type)
SkillGraph(user_id, topic, score)

(Your PDF already included these tables and the idea to record hints and progress).

13) Quick starter checklist (what to build first — day 0 → week 4)

1. Build Chrome extension skeleton that injects a simple panel on LeetCode and reads the editor contents.
2. Hook tree-sitter (WASM) in the content script and show syntax errors inline.
3. Create a simple Spring Boot API and a DB schema for Users+Attempts.
4. Add a rule-based hint engine: map compiler errors → curated hints stored in DB.

5. Wire one problem end-to-end (capture → send → backend → return hint → show).

Final notes & recommendation

- Start with **rule-based + local parsing** for a fast, demonstrable MVP (Semester 1 deliverable in your doc). Then stage-in LLMs and the skill graph in Semester 2. This approach matches your Major Project plan and gives strong, testable milestones.

- If you want, I can now: (pick one, I'll produce it right away)

    - Draft the exact **WebSocket message schema** (client↔server) for real-time hinting.

    - Give a **starter content script** (Chrome extension) that hooks LeetCode's editor and runs tree-sitter to surface syntax errors.

    - Design the **database SQL schema** and sample API endpoints for Attempts/Hints/SkillGraph.