International Journal of Computer Science and Engineering Development
Volume X, Issue X | Month Year | www.ijcsed.com
ISSN: 3139-0862

CodeRealm: Adaptive Gamified Learning and Synthetic Tutor

Kaivalya Narvekar, Prof. Dipti Mukadam, Sneha Shelar, Isha Tandel, Pranav Walunj

Department of Computer Engineering, Watumull Institute of Engineering and Technology
University of Mumbai, Mumbai, India

Abstract*—Programming learners often encounter conceptual difficulties that are not adequately addressed by compiler errors or conventional code-completion tools. On-demand conversational assistants can provide explanations, but they require learners to recognise that help is needed and formulate a question; unrestricted code generation may also reveal complete solutions before the underlying concept is understood. This paper presents CodeRealm, a proposed browser-based programming environment that initiates limited assistance from observable editor activity. The system combines inactivity duration, failed executions, repeated error categories, and recent code deletions through an interpretable weighted scoring rule. When the score crosses a configurable threshold and a cooldown condition is satisfied, a backend service constructs a constrained prompt for a locally hosted language model through Ollama. The model is asked to return structured conceptual feedback, a Socratic guiding question, a concept tag, and a Mermaid-compatible flowchart rather than a complete corrected program. A concept-mapped practice item then reinforces the same topic, while rule-based difficulty adjustment uses recent answers and attempts to select the next item. The paper describes the architecture, algorithms, implementation plan, and evaluation protocol. Since controlled empirical testing has not yet been completed, no measured learning gains or system-performance results are claimed.

***Keywords*—programming education; intelligent tutoring systems; programming struggle detection; Socratic feedback; large language models; local inference; gamified learning; adaptive practice**

I. Introduction

Learning to program involves more than memorising syntax. Students must form mental models of variables, control flow, data structures, functions, and program state. A compiler can identify many syntax and type errors, but it is less effective at explaining why a loop executes one extra time, why a condition selects the wrong branch, or why a function returns an unexpected value. These conceptual difficulties may persist even when the program produces an error message that is technically correct.

Conversational coding assistants provide another source of support, but they are commonly activated only after a learner submits a prompt. This places the burden of recognising difficulty and describing it on the student. In addition, a response that supplies a complete solution can shorten the immediate task while reducing the opportunity to practise reasoning. Research on guarded programming assistants and Socratic coding support indicates the value of assistance that guides learners without simply replacing their work [1], [2].

CodeRealm explores a complementary approach: assistance is considered proactively from coding-session signals, while the language model is constrained to explain and guide rather than to state complete solutions. The system does not claim to diagnose confusion directly. Instead, it estimates whether a moment may warrant support by combining observable events and allowing the learner to dismiss or ignore an intervention.

The proposed contributions are: (1) an interpretable, multi-signal rule for identifying potential moments of programming difficulty; (2) a local-LLM tutoring pipeline that returns structured conceptual hints and guiding questions; (3) visual explanations generated as Mermaid flowcharts; (4) concept-linked micro-practice with rule-based difficulty adjustment; and (5) an evaluation protocol covering trigger quality, response latency, output validity, and hint restraint.

II. Related Work

A. Hint-Based AI Assistance

Liffiton et al. describe CodeHelp, an LLM-powered support tool that uses guardrails to assist programming students without directly revealing solutions [1]. Their work motivates the use of constrained prompting and educationally appropriate responses. Gupta et al. present Sakshm AI, which combines Socratic tutoring and structured feedback for programming learners [2]. CodeRealm shares the goal of guided assistance but differs in its proposed trigger mechanism: it considers editor activity before a student explicitly submits a help request.

B. Proactive Assistance and Interaction Traces

Proactive assistance raises a timing problem: an intervention can be useful when a learner is stuck but distracting when the learner is reading, planning, or testing an idea. For this reason, CodeRealm does not treat a pause as sufficient evidence of difficulty. Work by Tabarsi et al. investigates how interaction traces in open-ended block-based programming projects relate to learner struggle [3]. Their findings support examining combinations of behavioural signals, while also reminding designers that traces are indicators rather than direct measurements of cognition.

 ISSN: 3139-0862

C. Generative AI and Learning

Studies of generative AI in programming education have raised questions about the balance between immediate task completion and durable understanding. CodeRealm responds through a tutoring policy that asks for concept-level feedback, a small hint, and a guiding question. The model is not authorised by the system design to return a complete corrected solution as its normal response. This is a design constraint that must be checked during evaluation rather than assumed to be perfectly enforced.

D. Gamification and Micro-Learning

A meta-analysis of gamification in programming education reports positive effects across outcomes such as motivation and academic achievement, while noting that design choices influence the result [4]. CodeRealm uses game elements as lightweight reinforcement for completing focused practice items. Points or streaks are secondary to the learning objective; they are not treated as evidence that a concept has been mastered.

E. Research Gap

The reviewed work provides evidence for guarded LLM assistance, Socratic tutoring, behavioural indicators of struggle, and gamified programming practice. CodeRealm investigates their integration in a single workflow in which editor signals initiate a constrained local-model response, followed by concept-matched practice. This is a scoped design contribution, not a claim that no prior system offers proactive coding support. A broader systematic comparison is required before making an exclusive novelty claim.

III. System Architecture

CodeRealm is organised into a browser-based interaction layer, a Django REST backend, a local language-model service, and a relational data store. The student uses a React interface containing Monaco Editor. An editor-behaviour component records relevant events such as text changes, run attempts, error categories, and deletions. When the rule-based detector decides that assistance may be useful, the frontend sends the task context and selected activity signals to the backend.

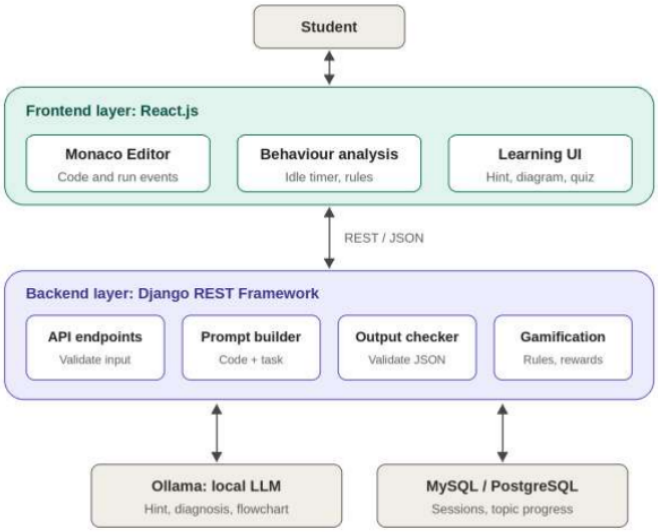


Fig. 1. Original CodeRealm system architecture and data flow (reproduced from the authors' initial project draft).

The Django REST service validates the request, constructs a restricted prompt, and communicates with the Ollama server hosting the selected local model. The response is parsed and checked before it is returned to the browser. Mermaid.js renders the diagram definition, while the learning interface presents the hint and the corresponding practice item. Session events, intervention decisions, and practice outcomes are stored in MySQL or PostgreSQL; the final database choice is an implementation decision.

A deliberate separation is maintained between deciding when to intervene and deciding what to say. The activity detector uses deterministic rules and does not depend on an LLM response. The LLM is responsible for producing educational content after a trigger has been accepted. This separation makes trigger behaviour measurable and prevents an unpredictable model response from controlling the timing of pop-ups.

IV. Proposed Methodology

A. Editor Event Collection and Idle Detection

The editor component listens for text-change events. Each relevant change resets an inactivity timer. If no new change occurs within the configured interval T_{idle}, the component records a potential idle event. The initial prototype value is T_{idle} = 15 s; this is a starting configuration, not a universally valid threshold. A cooldown interval $T_{cooldown}$ is applied after an intervention to avoid repeatedly interrupting the same work episode.

An idle event alone does not trigger an AI request. It causes the detector to collect additional signals for the current task. The interface should also allow a learner to dismiss an intervention and should avoid collecting unnecessary personal information. The signals retained for evaluation are limited to session identifiers, topic labels, event counts, trigger decisions, hint usage, and practice outcomes.

B. Multi-Signal Struggle Score

Let I be a binary inactivity indicator, F the number of failed runs within a defined observation window, E the count of repeated occurrences of the same error category, and D a normalized measure of recently deleted code. The proposed score is:

$$S = w_1 I + w_2 F' + w_3 E' + w_4 D' \quad (1)$$

TABLE I. Signals Used In The Struggle Score

Signal	Operational definition	Purpose
Idle (I)	No text change for T_{idle}	Candidate pause indicator
Failed runs (F')	Bounded count of failed executions in a window	Repeated unsuccessful attempts
Repeat error (E')	Frequency of the same error category	Persistent error pattern
Deletion (D')	Normalized amount of recent code deletion	Possible revision difficulty

where F', E', and D' are bounded or normalized values so that signals with different numeric ranges do not dominate merely because of their scale. The weights $w_1 \ldots w_4$ and the threshold S_{min} are configuration parameters. They must be selected and tuned using pilot observations; the paper does not assume that any particular weight set is optimal. An assistance request is eligible only when $S \geq S_{min}$ and the cooldown has expired.

The detector records the component values alongside each decision. This supports later inspection of false positives and false negatives and makes the rule easier to explain than an opaque classifier. A high score indicates a possible need for help, not proof that the learner is confused.

C. Trigger and Cooldown Decision

The decision stage checks three conditions: (i) a candidate event has been observed; (ii) the weighted score reaches S_{min}; and (iii) the cooldown has expired. If any condition is false, the system continues monitoring. If all conditions are true, the frontend or backend creates an assistance request and records the trigger reason. After the intervention, the cooldown prevents repeated prompts until a new work period has begun.

D. Constrained Hint Generation

The backend prompt contains the programming language, task statement, relevant code excerpt, latest error output, and the concept suspected from the activity context. The prompt instructs the model to identify a likely conceptual issue, provide one concise hint, ask one guiding question, and avoid returning a complete corrected program. The expected response is structured JSON with the fields *conceptual_error*, *hint*, *guiding_question*, *mermaid_flowchart*, and *concept_tag*.

Django validates the response structure and checks for required fields before returning it. Invalid JSON or missing fields causes a controlled fallback, such as a plain-text hint or a request to regenerate within a limited retry policy. Mermaid text is validated by attempting a render; invalid diagram content is not displayed as a successful flowchart. Prompt constraints reduce the risk of solution leakage but do not guarantee it, so the output must be audited during testing.

E. Concept-Mapped Practice and Difficulty Adjustment

The *concept_tag* selects a short exercise from a curated activity bank. For example, a loop-related issue can map to iteration tracing or output prediction; a branching issue can map to condition selection; an array issue can map to index identification; and a function issue can map to return-value prediction. The first prototype uses authored activity templates rather than unrestricted generation, making relevance and correctness easier to inspect.

Difficulty changes through deterministic rules. After a correct response, the next item may move from concept recognition to output prediction, short-code correction, or ordering solution steps. After an incorrect response, the next item may be simplified and accompanied by an additional hint. The rule set can also retain the current level when performance is mixed. This is rule-based adaptation, not reinforcement learning or a trained learner classifier.

V. Proposed Algorithms

A. Algorithm 1: Candidate Struggle Detection

Input: editor events, execution outcomes, error categories, deletion statistics. *Output:* assistance-trigger decision.

1. Initialize idle timer and cooldown state for the active coding session.
2. On every relevant text-change event, reset the idle timer.
3. If the timer reaches T_{idle}, mark a candidate idle event.
4. If no candidate event exists, continue monitoring.
5. Collect failed-run, repeated-error, and recent-deletion signals for the observation window.
6. Normalize or bound the numeric signals and calculate S using (1).
7. If $S \geq S_{min}$ and the cooldown has expired, request assistance.
8. Otherwise, retain the event for logging and continue monitoring.
9. After an intervention, start the cooldown interval.

B. Algorithm 2: Structured LLM Hint Generation

Input: task, code context, error output, suspected concept. *Output:* validated tutoring response.

1. Construct a prompt containing only the context required for the current task.
2. Specify the Socratic response policy and the required JSON fields.
3. Send the prompt to the locally hosted model through Ollama.
4. Parse the model response as JSON and check all required fields.
5. Reject or safely fall back when the response is malformed or incomplete.
6. Validate the Mermaid definition before rendering it in the browser.
7. Return the validated hint, guiding question, concept tag, and optional diagram.

C. Algorithm 3: Concept-Based Practice Selection

Input: concept tag, current activity level, answer correctness, attempt count. *Output:* next practice item.

1. Map the concept tag to the corresponding activity bank.
2. Select an item at the current activity level.
3. If the answer is correct, increase difficulty by one permitted level.
4. If the answer is incorrect, decrease difficulty or retain the level and provide support.
5. Record the answer, attempts, selected item, and updated difficulty.
6. Use the updated state for the next practice selection.

 ISSN: 3139-0862

VI. Implementation Plan

The proposed frontend is developed using React.js with Monaco Editor for code entry and event collection. The backend uses Python and Django REST Framework to expose APIs, apply the trigger rules, validate model responses, and persist session information. Ollama hosts the local language model; the exact model variant and quantization will be selected according to available memory and measured response time. Mermaid.js renders flowchart definitions in the browser. MySQL or PostgreSQL will be used for persistent records, with the final choice based on deployment and integration requirements.

The implementation will be modular. Editor monitoring, trigger scoring, prompt construction, response validation, diagram rendering, activity selection, and progress storage will be tested separately before end-to-end integration. Code execution, if enabled in the prototype, must be isolated from the application server because arbitrary student code is untrusted. The research prototype should not execute submitted programs with unrestricted host permissions.

VII. Evaluation Methodology

A. Research Questions

RQ1: How accurately does the multi-signal rule identify moments that independent reviewers judge as requiring assistance?

RQ2: What is the response latency from an accepted trigger to the display of a validated hint?

RQ3: How often does the LLM return valid structured output and renderable flowcharts?

RQ4: How often do generated hints reveal a complete solution instead of providing limited guidance?

RQ5: How relevant are concept-mapped practice items to the identified programming topic?

B. Experimental Procedure and Metrics

A pilot study can use a set of introductory programming tasks covering loops, conditional statements, arrays, and functions. With appropriate consent, editor events and trigger decisions can be logged during volunteer sessions. Two reviewers can independently label sampled decision points as appropriate or inappropriate for assistance using a documented procedure. Trigger precision, recall, and F1-score can then be calculated against these labels.

TABLE II. Evaluation Measures

Evaluation area	Measurement	Reporting method
Trigger quality	Precision, recall, F1-score	Against reviewer labels
Response latency	Request-to-display time	Median and 95th percentile
Structured output	Valid JSON rate	Percentage of valid responses
Diagram quality	Render success and logic review	Render rate plus manual rubric
Hint restraint	Complete-solution leakage rate	Manual review of sampled outputs
Practice relevance	Concept-to-item alignment	Reviewer rubric / learner feedback

Latency should be measured on the actual test machine with the selected local model and configuration. Results should report the hardware, model version, prompt settings, and sample count. Diagram evaluation should distinguish syntactic render success from semantic correctness: a diagram may render while still representing the wrong logic. Hint restraint should be assessed with a predefined rubric that identifies whether the response gives away the full solution.

A later controlled study may compare learners using CodeRealm with a suitable comparison condition on equivalent programming tasks. Such a study would need an approved protocol, an adequate sample, baseline measures, and a defined learning outcome. Until that work is completed, claims should remain limited to the proposed design and component-level evaluation.

VIII. Results And Discussion

At the time of writing, the prototype has not completed the planned empirical evaluation. Accordingly, this paper does not report measured detection accuracy, latency, learning gains, or comparative superiority. The expected technical outputs are a working activity-monitoring pipeline, an interpretable trigger log, structured local-model responses, validated flowcharts, and concept-linked practice records. These outputs will establish whether the integrated workflow functions as intended; they will not, by themselves, establish improved learning outcomes.

The main design trade-off concerns intervention timing. A conservative threshold may reduce unwanted prompts but miss some moments when learners need help. A permissive threshold may increase recall while also interrupting students who are thinking or reading. The pilot evaluation is therefore needed to compare weight settings and thresholds rather than selecting them by intuition alone. A second trade-off concerns local inference: keeping code on the local machine can reduce dependence on external services, but it does not automatically provide security, privacy, or reliable output. Access control, process isolation, data minimisation, and response validation remain necessary.

IX. Limitations And Future Work

The proposed struggle score is a heuristic and may misclassify normal pauses, exploratory edits, or deliberate code deletion as difficulty. Its weights and threshold require empirical tuning and may vary across tasks and learners. The local language model can produce incorrect explanations, malformed JSON, invalid diagrams, or responses that disclose too much of a solution. Output validation and human review reduce but do not eliminate these risks. The initial activity bank also covers a limited set of programming concepts, and the proposed adaptation is based on explicit rules rather than a learned student model.

Future work includes support for additional languages and concepts, retrieval grounded in instructor-approved course material, a teacher dashboard, improved privacy controls, and a larger learner study. If sufficient labelled interaction data becomes available, supervised models could be investigated as an alternative to hand-tuned scoring. Reinforcement learning is a possible research direction for activity sequencing, but it is outside the current system design and should not be presented as an implemented feature.

X. Conclusion

This paper presented CodeRealm, a proposed proactive programming tutor that combines editor activity signals with locally hosted language-model assistance. The design uses an interpretable weighted rule to identify candidate moments for support, then requests a constrained response containing a conceptual hint, a Socratic question, and an optional flowchart. A concept-mapped practice item follows, with difficulty adjusted through deterministic rules based on learner performance. By separating the trigger decision from language generation, the architecture supports independent testing of intervention timing and response quality. The next step is to implement the prototype, conduct pilot sessions, tune the scoring parameters, and report measured results; any claim about learning effectiveness must be established through a future controlled evaluation.

XI. Design Details And Operational Considerations

A. Observation Window and Signal Normalisation

The scoring process operates over a configurable observation window W rather than accumulating every event from the beginning of a session. This limits the influence of old failures that no longer reflect the learner's current progress. Failed executions and repeated error categories are counted within W, while recent deletions are measured over the same window or over a shorter edit-history interval. The implementation should record timestamps for each event so that the window can be reconstructed during evaluation.

Raw signal values have different scales. The idle indicator is binary, whereas failed runs and repeated errors are counts and deleted code may be measured in characters or lines. Directly adding these values can make the score depend on units rather than behavioural importance. Therefore, the count-based signals should be clipped at configurable maxima and mapped to a bounded range, for example by dividing by the corresponding cap and limiting the result to one. The exact transformation must be fixed before a pilot comparison so that competing weight configurations are evaluated consistently.

B. Weight Selection and Threshold Calibration

The initial weights are design parameters and should not be described as learned coefficients. A pilot dataset can be used to compare a small number of interpretable configurations. Reviewers label sampled decision points independently, and the system records the score and component values for each point. Candidate thresholds can then be evaluated using precision and recall. A high threshold generally reduces unnecessary interruptions but may miss genuine support opportunities; a low threshold may detect more candidate moments while increasing false prompts. The selected operating point should reflect the intended balance and be reported together with the observed error rates.

To avoid tuning and evaluating on exactly the same observations, the pilot should be divided by student or session into calibration and hold-out subsets where sample size permits. If the dataset is too small for a reliable split, the paper should describe the results as exploratory and avoid presenting them as generalisable. The rule should also be checked across different task types because a long pause during a recursion problem may have a different meaning from a pause during a short syntax exercise.

C. Intervention Policy and Learner Control

Proactive support can interrupt concentration if it is delivered too frequently. CodeRealm therefore uses a cooldown period after an intervention and should provide a clear dismiss or defer option. A dismissed hint should not be treated as proof that the trigger was incorrect; the learner may simply prefer to continue independently. Dismissal can be logged as a usability signal, but it should be interpreted separately from reviewer-labelled trigger quality. The interface should avoid repeatedly displaying the same hint when the learner has already viewed it.

The intervention should be concise and should not cover the code editor or obscure the task statement. A learner should be able to request additional explanation voluntarily. This preserves the proactive nature of the system without removing learner control. The initial prototype can use one hint at a time, followed by a single concept-matched activity, rather than presenting a long sequence of unsolicited messages.

D. Response Schema and Validation

The proposed response schema separates the model's diagnostic content from its learner-facing guidance. The *conceptual_error* field describes the likely underlying concept; *hint* contains a short nudge; *guiding_question* contains one question intended to prompt reasoning; *mermaid_flowchart* contains a diagram definition; and *concept_tag* identifies the activity-bank category. Each field should have length limits and type checks. Empty strings, unexpected keys, malformed JSON, or an unknown concept tag should be handled explicitly.

A syntactically valid response is not necessarily pedagogically correct. A flowchart can parse successfully while depicting an incorrect branch or loop condition. The evaluation should therefore measure both machine validation and human-rated semantic correctness. The backend should also avoid displaying model-generated code blocks as a default response. If the model violates the hint-only instruction, the response can be rejected, regenerated under a limited retry policy, or replaced with a safe fallback.

E. Data Handling and Execution Safety

CodeRealm processes behavioural data and submitted code, both of which should be treated as potentially sensitive. The prototype should collect only the fields needed for the stated evaluation, use session identifiers rather than unnecessary personal identifiers in research logs, and define a retention period. Local inference reduces the need to transmit source code to a third-party model service, but it does not by itself guarantee confidentiality. Access controls, secure storage, transport protection between application components, and clear participant information remain necessary.

If the platform executes student programs, execution must be isolated from the web application process. Submitted code is untrusted input and should run with resource limits, timeouts, restricted filesystem access, and no unnecessary network permissions. This execution-sandbox requirement is separate from the AI tutoring pipeline and should be implemented before the system is exposed beyond a controlled development environment.

XII. Evaluation Definitions

A. Trigger Metrics

Let TP denote moments labelled by reviewers as appropriate for assistance that the detector also triggers; FP denote triggered moments labelled as not requiring assistance; and FN denote reviewer-labelled moments for which the detector does not trigger. Trigger precision is TP/(TP+FP), recall is TP/(TP+FN), and F1 is the harmonic mean of precision and recall. These metrics should be reported with the number of labelled moments and, where feasible, confidence intervals. Reviewer agreement should also be reported because the reference labels involve judgement.

B. Latency and Output Metrics

End-to-end response latency is measured from the time the backend accepts an assistance request to the time the validated hint becomes visible in the browser. The median and 95th percentile should be reported because local model inference can produce a skewed latency distribution. JSON validity is the proportion of model responses that pass schema validation. Diagram render rate is the proportion of diagram definitions that render successfully; semantic diagram correctness requires a separate reviewer assessment.

C. Hint Restraint and Practice Relevance

A hint is considered to leak a complete solution when it supplies a complete working program or discloses the essential sequence of implementation steps in a way that removes the intended reasoning task. The evaluation rubric should distinguish a useful conceptual hint, a partial implementation clue, and a full solution. Practice relevance can be assessed by asking reviewers whether the selected activity tests the concept represented by the concept tag and whether the difficulty change is consistent with the learner's performance. These definitions should be fixed before analysing the sample.

XIII. Threats To Validity

Several threats may affect the interpretation of a future evaluation. Construct validity is limited because editor pauses and code changes are only indirect indicators of cognitive difficulty. Internal validity may be affected by task complexity, prior programming experience, interruptions outside the editor, and differences in hardware. External validity may be limited if the pilot uses a small volunteer sample or only introductory programming tasks. Model version, quantization, prompt wording, and local hardware can also change response quality and latency. These factors should be documented so that the evaluation can be reproduced.

A component-level pilot can show whether the workflow operates and whether triggers are reasonable, but not whether the intervention improves learning. To make a learning-effectiveness claim, a separate study should measure learning outcomes using comparable pre- and post-task assessments or a controlled comparison. Engagement indicators such as points, streaks, or activity completion should not be treated as substitutes for conceptual learning.

XIV. End-to-end System Workflow

The end-to-end workflow begins when a learner opens a programming task and starts editing in Monaco Editor. The frontend assigns a session identifier and associates the task with a topic label from the learning content. Relevant editor events are timestamped locally. The event collector maintains the current inactivity timer and a bounded summary of recent execution outcomes, error categories, and deletion activity. Only a compact feature summary is sent for trigger evaluation; the complete editor history is not required for every decision.

When the idle timer expires, the detector evaluates the current observation window. If the score remains below the configured threshold, the learner continues working without interruption. If the score reaches the threshold but the cooldown is active, the event is logged and no new prompt is displayed. When both the score and cooldown conditions permit an intervention, the backend receives the task context and relevant signals. The backend then constructs the model prompt, requests a structured response from Ollama, validates the response, and returns the accepted fields to the frontend.

The frontend presents the conceptual hint and guiding question in a non-blocking assistance panel. If the Mermaid definition passes validation and renders, the corresponding flowchart is displayed alongside the hint. The concept tag is

 ISSN: 3139-0862

used to retrieve a matching practice item from the activity bank. After the learner submits an answer, the activity module records correctness and attempts, updates the difficulty state using the configured rules, and stores the outcome for the session dashboard. This sequence creates a traceable link between the observed signal, the decision to intervene, the generated guidance, and the subsequent learning activity.

XV. Data Model And Module Responsibilities

A minimal relational data model may include Student, Session, Task, ActivityEvent, AssistanceEvent, PracticeItem, and PracticeAttempt entities. Student stores the account identifier and non-sensitive profile fields required by the application. Session records the start and end of a learning session and its selected task. ActivityEvent stores timestamped event categories or aggregated counts. AssistanceEvent stores the trigger score, threshold configuration, response status, and hint metadata. PracticeAttempt stores the selected concept, item difficulty, answer outcome, and attempt count. The model should avoid storing raw keystrokes unless a specific research question requires them and participant consent covers their collection.

The activity-monitoring module is responsible for collecting editor events and maintaining timers. The difficulty-detection module computes the weighted score and records the reason for each decision. The AI orchestration module builds prompts, calls Ollama, and validates the structured response. The visualization module validates and renders Mermaid definitions. The microlearning module maps concept tags to authored activities and applies difficulty rules. The analytics module aggregates session-level information for progress views and research evaluation. Keeping these responsibilities separate supports unit testing and makes failures easier to localise.

XVI. Reproducibility And Configuration

A reproducible evaluation should report the idle threshold, cooldown duration, observation-window length, score weights, score threshold, signal-normalisation caps, model name and version, model quantization where applicable, Ollama version, prompt template version, and test hardware. These settings can materially affect the trigger rate and response latency. The system should store a configuration identifier with each evaluated session so that results from different settings are not unintentionally combined.

For each experiment, the same task instructions and activity-bank items should be used across compared configurations. Reviewers should receive a written labelling guide with examples of appropriate and inappropriate assistance moments. Where feasible, reviewers should be blinded to the system's decision and score when assigning labels to reduce bias. Disagreements can be reported using percentage agreement or an appropriate inter-rater statistic. These procedures make the trigger evaluation more transparent and easier for another group to reproduce.

XVII. Ethical And Educational Considerations

The system is intended to support learning rather than monitor students for disciplinary purposes. Participation in any user study should be voluntary, and students should be informed about what activity information is collected, why it is collected, how long it is retained, and who can access it. The interface should make clear that a trigger is a possible support opportunity, not a judgement of ability or effort. Students should be able to continue coding without accepting a hint.

AI-generated guidance can be inaccurate or overly confident. The platform should present hints as assistance rather than authoritative grading, and students should retain access to course materials and instructor support. The activity bank should be reviewed for correctness and alignment with the course. If the platform is used with minors or in a formal educational setting, institutional ethics and data-protection requirements must be followed before collecting research data.

References

[1] M. Liffiton, B. E. Sheese, J. Savelka, and P. Denny, "CodeHelp: Using large language models with guardrails for scalable support in programming classes," arXiv:2308.06921, 2023.

[2] R. Gupta, H. Goyal, D. Kumar, A. Mehra, S. Sharma, K. Mittal, J. S. Challa, *et al.*, "Sakshm AI: Advancing AI-assisted coding education for engineering students in India through Socratic tutoring and comprehensive feedback," arXiv:2503.12479, 2025. Published version: *ACM J. Comput. Sustain. Soc.*, 2026, doi: 10.1145/3788679.

[3] B. T. Tabarsi, A. Limke, H. Reichert, R. Qualls, T. Price, C. Martens, and T. Barnes, "How to catch novice programmers' struggles: Detecting moments of struggle in open-ended block-based programming projects using trace log data," in *Proc. 6th Educ. Data Mining Comput. Sci. Educ. (CSEDM) Workshop*, Durham, U.K., 2022, doi: 10.5281/zenodo.6983260.

[4] Z. Zhan, L. He, Y. Tong, X. Liang, S. Guo, and X. Lan, "The effectiveness of gamification in programming education: Evidence from a meta-analysis," *Comput. Educ.: Artif. Intell.*, vol. 3, Art. no. 100096, 2022, doi: 10.1016/j.caeai.2022.100096.

[5] "Django REST framework." [Online]. Available: https://www.django-rest-framework.org/

[6] "Monaco Editor." [Online]. Available: https://github.com/microsoft/monaco-editor

[7] "Mermaid." [Online]. Available: https://mermaid.js.org/

[8] "CodeAid: Evaluating a classroom deployment of an LLM-based programming assistant," 2024.

[9] "Level up your coding: A systematic review of personalized, cognitive, and gamified learning in programming education," 2024.

[10] "SocraticLM: Exploring Socratic personalized teaching with large language models," 2024.

[11] "Evaluating the application of large language models to generate feedback in programming education," 2024.

[12] "Evaluating the effectiveness of LLMs in introductory computer science education: A semester-long field study," 2024.

[13] "The impact of large language models on programming education and student learning outcomes," 2024.

[14] "Adaptive architectures for gamified learning in software engineering: A systematic review," 2026.

[15] "Artificial intelligence in programming education: A systematic review of intelligent, adaptive, and generative systems, learner outcomes, and implementation challenges," 2026.