Trends and Reviews of Technological Engineering in Education
Vol. 2, No. 2 (2026) | ISSN 3071-2211 | Open Access

RESEARCH ARTICLE

Tool-Integrated Large Language Models for Stepwise Debugging and Feedback in Engineering Programming Laboratories

Zhipeng Luo[1], Yanfei Xiong[2,*]
[1] School of Intelligent Manufacturing, Guangzhou Huali College, Guangzhou, China 511325
[2] School of Intelligent Manufacturing, Jiangxi University of Applied Science and Technology, Nanchang, China 330100
[*] Correspondence: 240559079@qq.com

Received 20 December 2025 · Accepted 28 April 2026 · Published 30 June 2026

Abstract

Engineering programming laboratories require students to interpret compiler output, form hypotheses, inspect program behavior, and revise code while instructors provide timely guidance. Conventional diagnostics can be precise yet difficult for novices to translate into a next action; unconstrained language-model responses can be fluent yet unsupported, overly complete, or inconsistent with a laboratory's learning goals. This article specifies a tool-integrated large language model (LLM) architecture that converts authorized compiler, test, static-analysis, and debugger evidence into staged, evidence-linked feedback. A sandboxed execution broker runs only instructor-approved tools and records immutable run snapshots. A diagnostic normalizer maps raw outputs to typed evidence objects. A structured response schema binds each observation or interpretation to evidence identifiers. Deterministic validators check schema, identifier existence, snapshot and run freshness, source integrity, and policy limits; they do not by themselves prove semantic entailment. Semantic support is treated as a separate evaluation question. The LLM may propose a qualified fault hypothesis and a hint at a policy-selected level, but cannot change code, grades, or exercise tests. A synthetic heat-conduction exercise illustrates the data flow and learner interaction. The article contributes a reference architecture, event and claim-level evidence schemas, a stepwise feedback protocol, security controls, a design traceability method, and a prospective verification plan. It reports no classroom deployment, participant data, or learning effects; these remain questions for implementation and future evaluation.

Keywords: Engineering Programming Education; Large Language Models; Program Debugging; Execution Feedback; Formative Assessment; Learning Scaffolds; Secure Code Execution

1. Introduction

Programming laboratories are central to engineering education because students use computation to model systems, analyze measurements, automate calculations, and test design decisions. The work is rarely limited to writing a syntactically valid program. A learner must connect a physical or mathematical model to variables and data structures, predict behavior, observe what the program actually does, and use discrepancies to decide what to inspect next. Debugging is therefore both a software practice and a disciplinary reasoning activity. In a laboratory, a student may need help with a compiler error, an indexing fault, a unit conversion, or an assumption embedded in a numerical model; each problem calls for a different kind of evidence and guidance.

The assistance bottleneck is familiar. Compiler and test messages can identify a failed check but do not always explain why the failure matters or what a novice should inspect. Instructors and teaching assistants can interpret those signals, but they may not be available at the moment a learner encounters an error. Automated assessment systems can scale correctness checks, while earlier feedback systems have explored error localization, test-based feedback, and generated hints. Their design illustrates a continuing trade-off: feedback that is too vague does not support action, while feedback that supplies a complete repair may bypass the learner's own diagnosis and explanation [1], [2].

Large language models can translate technical outputs into conversational explanations, compare possible causes, and adapt the form of a hint to an instructional policy. They also introduce a distinct reliability problem. A model can misread a trace, infer a defect that is not present, invent tool output, reveal a complete answer, or present a guess as a confirmed diagnosis. Code-generation benchmarks and self-debugging methods demonstrate the technical possibility of coupling language models with execution, but a model's ability to improve generated code does not itself define a safe teaching workflow [3], [4], [5], [6]. Educational feedback must be grounded in the learner's actual run and designed to preserve the learner's role in making and testing a change.

© 2026 TRTEE. Open Access under CC BY 4.0

This article specifies a tool-integrated LLM system for engineering programming laboratories. Compiler, test, static-analysis, and debugger tools produce bounded execution evidence; the language model is a constrained interpreter and hint drafter. It cannot execute arbitrary tools, edit a learner's submission, determine a grade, or declare a diagnosis verified. A pedagogical controller selects the maximum allowed hint level. Claim-level evidence links and deterministic freshness and policy checks constrain what can be presented, while semantic support is not treated as mechanically proven by the presence of an evidence identifier. Learners remain responsible for changing code, rerunning it, and explaining the result.

The manuscript makes six design contributions: (1) a reference architecture that separates execution authority from model-generated explanation; (2) typed records for code snapshots, tool runs, evidence, claim segments, hypotheses, hints, and learner actions; (3) a finite-state feedback protocol with configurable hint escalation; (4) a reproducible synthetic engineering fixture that follows a boundary-condition error from tool evidence to a learner-verified correction; (5) a requirement-to-component-to-test traceability method; and (6) a prospective technical, safety, and pedagogical verification plan. The proposal is intended for later implementation and evaluation. It is not a report of a deployed system, a classroom experiment, or a systematic review, and it makes no claims about learning gains or time savings.

2. Instructional and Technical Foundations

2.1 Debugging as an engineering learning activity

Debugging is not a single act of correcting a line. It includes recognizing that an observed result differs from an expected result, building a mental model of relevant code, locating a plausible cause, testing that hypothesis, and deciding whether the repair preserves intended behavior. Novice programmers may struggle to connect textual code with its execution, especially when loops, state changes, nested conditions, or unfamiliar error messages are involved. Research on novice errors and program comprehension documents that students can produce code that looks plausible while holding an inaccurate model of what it will do [7], [8], [9], [10], [11]. Interactive environments such as Ladebug make debugging actions

© 2026 TRTEE. Open Access under CC BY 4.0

and feedback an explicit instructional object [12]. A support system should therefore make the evidence and reasoning path visible, not merely return a corrected program.

Engineering laboratories add model-specific constraints. A program can run without an exception and still violate a boundary condition, use inconsistent units, mishandle missing measurements, or apply a numerical method outside its assumptions. A generic code assistant that sees only source text may not know which physical constraints are part of the exercise. The proposed design binds each interaction to an instructor-authored task contract containing expected inputs, permitted tools, test semantics, relevant concepts, and feedback policy. The LLM may use that context to explain a diagnostic, but the contract—not a conversational guess—defines what counts as a valid run or an allowed hint. Question-driven views of program behavior can help learners investigate causal claims, a useful interaction pattern for an evidence-oriented assistant [13].

Tool evidence and learner understanding are separate objects. A failing test is evidence that a particular input produced an output that differed from a declared expectation; it is not proof of the only possible defect. A debugger trace may show that a variable took an unexpected value at a program point; it does not establish why the value arose. The system records observations first and hypotheses second. The learner's revised submission and its subsequent run then provide new evidence. This separation supports epistemic discipline: the system can say what was observed, identify a plausible explanation, and ask the learner to test it without collapsing those steps into an authoritative answer.

2.2 Feedback, scaffolding, and learner agency

Formative feedback is useful when it helps learners understand a target, compare current work with that target, and choose a productive next action. Feedback effects vary with the information supplied and how the learner can use it; a message that merely evaluates correctness may be less instructive than one that helps a learner close a specific gap [14], [15], [16]. The system therefore generates action-oriented prompts tied to a concrete diagnostic, such as inspecting an array boundary or comparing two intermediate values. Each message has a limited instructional purpose and an explicit evidence link.

Scaffolding is contingent support: the assistance responds to a learner's present need and

© 2026 TRTEE. Open Access under CC BY 4.0

can be reduced as the learner takes more control. Wood, Bruner, and Ross [17] described tutoring support that helps a learner manage aspects of a task beyond current independent performance; later work on intelligent tutoring systems likewise emphasizes the structure and timing of guidance [18]. In the proposed workflow, the first response is deliberately small. The learner can ask for another hint, submit a hypothesis, or request an explanation. The controller advances only within the instructor's ceiling and records the reason for escalation.

This design does not assume that withholding information is always beneficial. A learner may need a direct explanation because of accessibility, a time-critical laboratory, or a task policy that permits worked examples. The interface exposes the current support mode and available actions, and the instructor can allow or disallow partial code, pseudocode, or a complete worked solution. The default sequence favors diagnosis before repair, but it is configurable and must be stated before the exercise begins. Learner agency includes the ability to request help and understand the consequences of a hint level, not merely being forced to struggle.

2.3 Why combine language models with deterministic tools

A compiler, unit test, linter, or debugger produces a narrow output under defined operational semantics. A language model produces a flexible explanation conditioned on supplied context. Their strengths are complementary only when their authority is kept distinct. For example, a test runner may report that the final temperature node is 0.0 °C rather than 25.0 °C. A trace may show that the right boundary value was written to index n-2. The model can suggest inspecting the assignment target, but the explanation remains a hypothesis until a new run supplies further evidence.

Tool use and execution-guided revision are explored in [4], [19], [20]. Self-Refine uses the same LLM to generate feedback and revise its output [21]; that self-feedback is not a compiler or debugger observation. Earlier synthesis-based feedback systems derive repairs using reference implementations and explicit error models [22]. Research on code generation and LLM-authored programming exercises illustrates technical opportunities, but does not by itself define a classroom feedback policy [23], [24], [25].

© 2026 TRTEE. Open Access under CC BY 4.0

Recent educational program-repair work also uses repair tasks to improve generated student feedback [26]. The present specification makes a different authority choice: the LLM does not apply repairs or select tools, and learners perform and test their own changes. Table 1 compares these adjacent design families on the dimensions relevant to this proposal. The comparison is selective and is not a systematic review.

Retrieval can provide approved explanations and task-specific definitions, but retrieved text is context, not execution evidence. The system includes only course-approved materials and preserves their versions and passage locators. A course explanation must cite an approved source; a run observation must cite current tool evidence. Missing or invalid references trigger a rewrite or safe fallback. These checks establish traceability, not semantic support. Unsupported interpretations require separate review or abstention. Retrieval-augmented generation can reduce dependence on ungrounded recall but does not guarantee factuality [27], [28].

2.4 Positioning among adjacent design approaches

The proposed system sits at the intersection of automated programming feedback, LLM-supported programming education, tool-use research, and formative-feedback design. Its contribution is the explicit combination of course-controlled tool execution, versioned evidence, claim-level feedback, and learner-performed repair. The comparison in Table 1 identifies this boundary without claiming that any one neighboring approach lacks all of these features.

Table 1. Design positioning against adjacent approaches

Approach family	Typical focus represented in cited work	Boundary addressed by this design
Automated programming feedback	Compiler, testing, localization, and feedback-generation methods [1], [2].	Adds a course-governed LLM explanation layer while retaining deterministic tools as the source of execution observations.
LLM-supported programming education	Natural-language support, code generation, and educational program-repair feedback [23], [24], [25], [26].	Constrains the model from editing code or applying repairs; the learner performs and tests the change.
LLM tool use and self-debugging	Methods for tool selection, interleaved reasoning, or execution-guided model revision [4], [19], [20].	Uses tools as instructor-approved evidence producers; the model cannot invoke tools or conduct an autonomous repair loop.
Formative feedback and scaffolding	Principles for actionable feedback, learner agency, and contingent support [15], [16], [29].	Operationalizes these principles through task-specific hint ceilings, explicit learner actions, evidence labels, and instructor handoff.

© 2026 TRTEE. Open Access under CC BY 4.0

Approach family	Typical focus represented in cited work	Boundary addressed by this design
This design specification	A technical architecture for tool-mediated, stepwise debugging in engineering laboratories.	Combines snapshot-bound tool evidence, claim-level response segments, a canonical state protocol, and prospective release gates; it does not claim learning effects.

2.5 Design boundary and terminology

The term tool-integrated refers to a controlled pipeline that invokes approved development tools and uses their structured outputs in a feedback event. It does not mean that the LLM can browse a student's filesystem, install arbitrary packages, access the internet, or run unrestricted shell commands. The term stepwise refers to feedback that is staged across learner actions and evidence updates. It does not imply a fixed universal sequence for all students or tasks. The term verification refers to explicit checks of code behavior against declared test or task conditions, not a proof of program correctness for every input.

The design separates four kinds of statement in the user interface: observed fact, interpretation, suggested action, and policy restriction. For example, “E1 reports 0.0 °C at index 4; the expected value is 25.0 °C” is an observation; “the right boundary may be assigned to the wrong index” is an interpretation; “inspect the assignment destination” is an action; and “the assistant will not provide a full solution in this exercise” is a policy. These labels help learners distinguish measured output from the model's reasoning and give instructors an audit trail for reviewing whether the system overstates what a tool established.

3. Design Scope and System Requirements

3.1 Target use and assumptions

The target setting is a programming laboratory in an engineering course where students submit code to a controlled workspace and use it to solve a defined technical task. The exercise has an instructor-owned specification, a pinned runtime image, a set of permitted dependencies, one or more tests or diagnostic tools, a feedback policy, and an assessment policy. The model receives only the minimum code and context needed to draft a response. It does not receive credentials, unrelated course records, or a student's identity unless the institution has a justified and approved need for that information.

The primary unit of interaction is one feedback episode: a code snapshot is submitted,

© 2026 TRTEE. Open Access under CC BY 4.0

approved tools run against that exact snapshot, evidence is normalized, an optional hint is generated, and the learner either acts or requests additional support. The system may preserve a sequence of episodes for the same exercise, but it must not silently combine results from different code versions. This design assumes that the learning platform can associate a submission with a course and task identifier and that the institution can define retention, access, and appeal policies.

A course may use several languages and toolchains, but each task must declare the supported set. The initial implementation should favor reproducibility over unrestricted flexibility: each run is tied to a container or virtual-environment image digest, compiler/runtime versions, test-bundle version, tool-adapter versions, and resource limits. If an environment cannot be reproduced, the interface should disclose that the recorded result may not be directly comparable to a later run. These are design assumptions rather than claims that every engineering program can meet them; local feasibility is part of implementation planning.

3.2 Design approach and requirement traceability

The architecture was developed through a normative, problem-driven design synthesis rather than stakeholder interviews or classroom observation. The procedure has four steps: define the instructional and execution boundary; derive requirements from evidence integrity, learner agency, reproducibility, and governance; map each requirement to a component and data invariant; and specify a verification artifact for each requirement. Table 2 makes that reasoning traceable. It is a design rationale, not an empirical validation of the requirements.

This traceability method also limits the scope of the contribution. The paper specifies what a prototype must preserve and how it can be checked; it does not claim that the chosen controls are sufficient for every institution, language, or engineering task. Local teams must bind the design to their course policies, runtime environment, and accessibility requirements.

© 2026 TRTEE. Open Access under CC BY 4.0

Table 2. Design traceability from instructional and safety needs to tests

Design need	Requirement	Component or invariant	Verification artifact
Separate observations from explanations and preserve provenance.	R3–R4	Diagnostic, EvidenceSpan, ClaimSegment; current snapshot/run links.	Stale-ID, altered-excerpt, ambiguous-evidence, and claim-support fixtures.
Keep the learner responsible for diagnosis and code changes.	R5–R6	Hint controller, disclosure ceiling, LearnerAction; no model edit permission.	Over-ceiling, auto-edit, explicit-request, and handoff workflow tests.
Make execution reproducible and bounded.	R1–R2	Immutable snapshot, environment manifest, allowlisted broker.	File-mutation, unauthorized-tool, timeout, resource-limit, and replay tests.
Fail safely and protect student information.	R7–R8	SafeFallback, append-only audit, role-based access, retention policy.	Malformed-response, outage, access-control, deletion, and threat-model review.

3.3 System requirements

The requirements below define observable design obligations for a prototype. They are not claims that the system has been built or passed these checks. Requirements R1–R4 protect execution and evidence integrity; R5–R8 govern instructional behavior, failure handling, and safety. Table 2 traces the design needs to requirements, components, and verification artifacts; Table 3 states the requirements in testable form.

Table 3. System requirements and verification evidence

ID	Requirement	System behavior	Verification evidence
R1	Snapshot integrity	Every tool result and hint is bound to an immutable code digest and task version.	Mutation and referential-integrity tests.
R2	Tool authority	Only instructor-approved adapters run, with fixed inputs and bounded resources.	Allowlist, timeout, and adversarial execution tests.
R3	Evidence provenance	Observations retain producer, version, run, source locator, and raw record.	Schema and source-resolution tests.
R4	Claim-level grounding	Each segment links to current evidence or an approved course source; linkage checks do not prove semantic entailment.	Reference/freshness fixtures plus separate human-rated semantic-support cases.
R5	Staged disclosure	Hint level cannot exceed published course policy; escalation follows learner action.	Workflow and policy-boundary tests.
R6	Learner agency	The system does not edit student code, assign grades, or silently submit a repair.	Permission tests and interaction review.
R7	Safe failure	Invalid model output falls back to a neutral diagnostic or instructor handoff.	Malformed output and outage tests.

© 2026 TRTEE. Open Access under CC BY 4.0

ID	Requirement	System behavior	Verification evidence
R8	Privacy and security	Execution and prompts follow data-minimization, access, and retention rules.	Threat-model review and penetration tests.

3.4 Non-goals and authority boundaries

The system is not an autonomous grader, code generator that completes a learner's assignment, plagiarism detector, or substitute for an instructor. It does not infer a student's ability from a single error or use hidden behavioral profiling to set a grade. A test pass is not equivalent to mastery, and a test failure is not a complete explanation of student understanding. If a course chooses to use automated tests in grading, the grading logic must be separately specified, published, and appealable; the feedback model is not authorized to change it.

The tool broker is the only component authorized to execute student code. The model can request no tool directly; it receives a bounded evidence packet after a run. If future implementations support model-initiated reruns, those must be mediated by a fixed allowlist and explicit budget, and each run must be visible in the learner and instructor audit trail. The baseline architecture intentionally avoids autonomous repeated execution because silent trial-and-error can consume resources and obscure which evidence informed a response.

4. Reference Architecture and Data Model

4.1 Components and data flow

The architecture consists of eight logical components: a course-policy registry, submission snapshot service, execution broker, tool adapters, diagnostic normalizer, append-only evidence/event store, feedback orchestrator, and learner/instructor interfaces. The registry binds an exercise to its language, runtime image, resource limits, permitted tool set, test semantics, hint ceiling, disclosure rules, approved retrieval collection, retention period, and escalation route. The submission service freezes relevant files as a content-addressed snapshot before execution. Every downstream object refers to the snapshot and task-policy version, so results cannot silently attach to later edits.

The execution broker starts an isolated run using the exercise's pinned environment and

© 2026 TRTEE. Open Access under CC BY 4.0

invokes only configured adapters. Adapters can include a compiler or interpreter, instructor-authored unit tests, static analyzer, formatter, and debugger configured for the exercise. The broker captures exit status, standard output and error, test identifiers, selected trace points, duration, and resource use. It rejects network access and writes outside a task workspace, subject to the host platform's sandbox capability. The broker records the actual tool versions rather than allowing the model to infer them from a prompt.

The normalizer converts raw tool outputs into typed diagnostics, evidence spans, and status codes. It preserves the original message and a normalized category, such as syntax, runtime, assertion, boundary, or resource-limit failure. The orchestrator constructs a minimal evidence packet and asks the model for a response conforming to a JSON schema. A validator checks schema, evidence references, policy constraints, and prohibited content. If validation succeeds, the interface displays the hint alongside its supporting evidence and learner options. Figure 1 shows the separation of authorities and the path of a feedback episode.

The evidence/event store is append-oriented. A new code version or tool run creates new records rather than overwriting prior evidence. Each event records the snapshot, run, task-contract version, policy version, actor, timestamp, and transition outcome. Instructors can inspect an episode while access controls limit who can see source code, traces, or learner annotations. Events use a pseudonymous learner key when identity is unnecessary. Model-provider data handling remains a separate operational review, including whether prompts or outputs are retained for training or service improvement.

© 2026 TRTEE. Open Access under CC BY 4.0

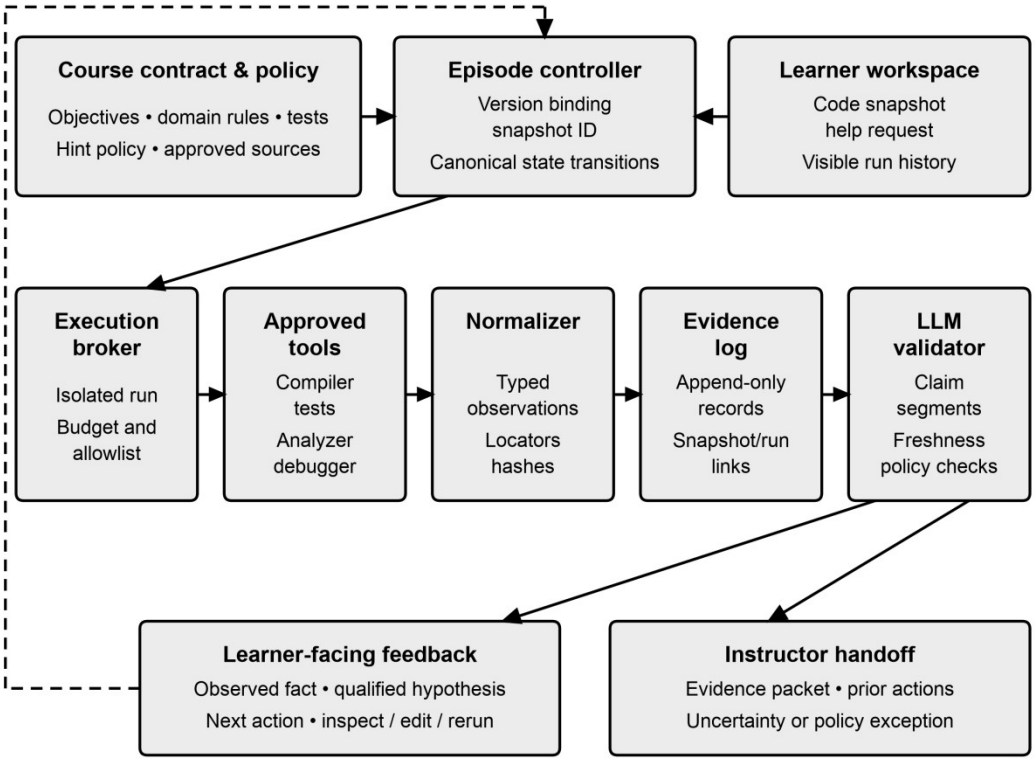

Figure 1. Evidence-to-Feedback Loop with Separated Authority

4.2 Tool adapters and evidence authority

A tool adapter has a declared input contract, output schema, version, timeout, resource budget, and failure behavior. An adapter cannot request arbitrary files or choose an unbounded command. The course author selects tools from an approved catalog and configures task-specific arguments. For instance, a Python lab may expose an interpreter run, unit test suite, static type check, and a debugger trace over preselected variables. An embedded systems lab may expose a simulator or static timing report rather than physical equipment. Tool selection remains discipline-specific.

Each evidence record includes evidence_id, snapshot_id, run_id, producer, producer_version, evidence_type, source_locator, observed_value, expected_value when configured, content hash, status, and timestamp. A source locator may identify a file and line range, test case, stack frame, variable at a trace step, or course-note passage. Raw output is

© 2026 TRTEE. Open Access under CC BY 4.0

retained in a restricted store; only a sanitized excerpt is available to the model. Normalization is loss-aware: when a parser cannot reliably interpret output, it marks the record unparsed and passes a safe excerpt rather than fabricating a structured diagnosis.

Tool evidence is authoritative only for the observation it directly reports. A compiler diagnostic can support a claim that a source expression failed compilation under a recorded compiler version. It does not establish that the compiler's message is pedagogically sufficient or identify the student's misconception. A unit test can support a claim about one input-output pair under one implementation and test bundle. It cannot establish correctness outside the tested domain. The response generator must use carefully scoped language—“this run shows,” “the test case indicates,” or “one possible cause”—and avoid broad claims such as “your algorithm is wrong” without supporting analysis.

The model's generated hypothesis is stored separately from observations and begins as unverified. A hypothesis may be marked supported only when a declared check produces evidence consistent with its prediction; it may be contradicted when a check produces evidence inconsistent with that prediction; otherwise it remains unresolved. These labels describe support within the declared fixture and test scope, not proof of a unique cause or general program correctness. A passing test after a code change does not alone establish that the proposed hypothesis caused the repair.

4.3 Typed interaction and evidence records

The minimum data model contains nine record types. SubmissionSnapshot identifies an immutable set of files and its digest. ExecutionRun records the environment manifest. Diagnostic preserves a normalized tool finding. EvidenceSpan identifies the exact output, test, trace, or approved source. ClaimSegment stores one learner-facing observation, interpretation, action, or policy statement with its evidence links and certainty label. Hypothesis records a proposed explanation and its state. Hint records the permitted support level and next action. LearnerAction captures a request, edit, annotation, or rerun. FeedbackEvent links these objects in order. Table 4 defines their minimum fields and invariants.

The response record preserves both machine-readable structure and student-facing

© 2026 TRTEE. Open Access under CC BY 4.0

language. Table 4 uses text for the wire field learner_facing_text; Hypothesis.claim_id refers to ClaimSegment.segment_id. The orchestrator adds source scope and validation status, and links suggested actions to Hint records with target concept and disclosure level. Observations cite current tool evidence; policy statements cite the active contract. Course-source spans identify approved versions, not tool observations. Hypotheses remain unverified until a declared check supports them. These links permit auditing; semantic support still requires separate review.

A task's learning contract is represented separately from a student's interaction. It contains learning objectives, domain assumptions, variables and units where relevant, boundary or operating conditions, test semantics, permitted hints, accessible alternatives, tool manifest, and grading-policy reference. The model receives a scoped projection of this contract, not the entire course database. A new test bundle, domain model, or hint policy creates a new contract revision; historical feedback stays associated with the exact revision that governed it.

Table 4. Typed records and evidence invariants

Record	Core fields	Invariant
SubmissionSnapshot	snapshot_id; task_version; policy_version; content_digest; file_manifest	Immutable; every run and response identifies this exact snapshot.
ExecutionRun	run_id; snapshot_id; image_digest; tool_versions; status; limits	One run uses one declared environment and bounded budget.
Diagnostic	diagnostic_id; producer; category; observed; expected; locator; content_hash	Tool observation is preserved separately from interpretation.
EvidenceSpan	evidence_id; run_id; source_type; source_locator; excerpt_digest	run_id is null for course/policy sources; source_locator binds their approved version.
ClaimSegment	segment_id; response_id; kind; text; evidence_ids; certainty; validation_status	Each observation, hypothesis, or policy claim has a typed status and traceable source.
Hypothesis	hypothesis_id; claim_id; supporting_ids; disconfirming_ids; rationale; state	Starts unverified; later states require a declared check and remain scoped.
Hint	hint_id; response_id; level; target_concept; next_action; disclosure	Level is within policy; the system does not apply the learner's repair.
LearnerAction	action_id; type; snapshot_id; request; annotation; timestamp	Learner action is timestamped and linked to the episode.
FeedbackEvent	event_id; prior_event; run_id; hint_id; policy_version; outcome	Events are append-only and ordered; stale responses are not presented.

© 2026 TRTEE. Open Access under CC BY 4.0

4.4 Model interface and output contract

The model receives a structured request containing the task objective, current hint ceiling, a sanitized code excerpt or authorized snapshot, selected diagnostic records, relevant approved course-note passages, and the learner's explicit question if any. It does not receive hidden test answers unless policy permits them, and it is not asked to infer a grade. The request states that tool outputs are observations, hypotheses must be labeled, and unsupported claims should be omitted. System prompts are versioned with the exercise and included in the build manifest.

The response requires response_id, snapshot_id, run_id, segments, next_action, hint_level, code_disclosure, and uncertainty. Each segment has segment_id, kind, learner_facing_text, evidence_ids or approved-source identifiers, and certainty. The orchestrator binds IDs to its trusted request; model-supplied IDs grant no authority. Validators check schema, reference existence, freshness, source hashes, and declared policy limits. A permitted hint_level or code_disclosure label does not establish that the actual text obeys it. Content disclosure and semantic support require separate evaluation and review; structural validity alone proves neither.

The following schematic object separates a tool observation from an unverified interpretation; identifiers are illustrative.

```
{
  "response_id": "resp-04", "snapshot_id": "snap-17",
  "run_id": "run-04", "hint_level": 1,
  "code_disclosure": "none",
  "uncertainty": "Hypothesis unverified.",
  "next_action": "Inspect the right-boundary index.",
  "segments": [
    {"segment_id": "c1", "kind": "observation",
     "learner_facing_text": "E1: index 4 is 0.0; expected 25.0.",
     "evidence_ids": ["E1"], "certainty": "observed"
    },
    {"segment_id": "c2", "kind": "hypothesis",
     "learner_facing_text": "The right boundary may use the wrong index.",
     "evidence_ids": ["E1", "E2"], "certainty": "unverified"
    },
    {"segment_id": "c3", "kind": "action",
     "learner_facing_text": "Inspect the right-boundary index.",
     "evidence_ids": ["E2"], "certainty": "suggested"
    }
  ]
}
```

Schema and policy failures share one repair attempt per request. If that attempt fails, the interface uses an authorized, sanitized diagnostic template or routes the episode to

© 2026 TRTEE. Open Access under CC BY 4.0

an instructor. Fallback never exposes unrestricted raw output: the same access, redaction, and disclosure rules apply. If no safe diagnostic exists, show only service status and a help route. A failed model call, tool timeout, or invalid response is neither task completion nor evidence that the learner's code is wrong. No unsupported cause is asserted.

5. Stepwise Diagnostic and Feedback Protocol

5.1 Feedback states

A feedback episode uses one state vocabulary: Ready, SnapshotFrozen, ToolsRunning, EvidenceNormalized, HintDrafted, ResponseValidated, AwaitingLearner, RecheckRunning, ResolvedForNow, InstructorEscalation, and SafeFallback. Transitions record the snapshot, run, contract, policy, and request. Table 5 specifies their guards and failure paths. ResolvedForNow records either all required checks passing in a completed run or the learner explicitly stopping. These closure reasons are stored separately; stopping and incomplete checks are not passes, and passing does not prove universal program correctness.

The controller freezes a submission and invokes only approved adapters. Before display, it atomically checks the response against the active snapshot, run, contract, policy, request, and open episode. A new run or superseding request can invalidate a response even when code is unchanged. Late callbacks cannot reopen a stopped or handed-off episode. Duplicate event IDs do not consume another hint or retry budget. An edit freezes a new snapshot; tool failures enter SafeFallback or InstructorEscalation. Current, usable evidence is required before any hint is drafted.

The controller distinguishes an exhausted hint budget from a technical failure. If a model response is unavailable, the learner may still inspect raw diagnostics, retry later, or contact the instructor. The event record identifies whether an episode ended because configured checks passed, the learner stopped, the service failed, or an instructor took over. Only the first two may close an episode as ResolvedForNow; a service outage must not be represented as task completion.

© 2026 TRTEE. Open Access under CC BY 4.0

5.2 Hint ladder and escalation rules

The proposed default ladder has five levels. Level 0 acknowledges the observed result and asks the learner to state the expected behavior. Level 1 points to a region or diagnostic category without naming a fix. Level 2 supplies a concept prompt, such as checking a loop's first and last valid index. Level 3 presents a short reasoning sequence or partial pseudocode. Level 4 may show a minimal code pattern or worked example only when the instructor has enabled that level and the learner explicitly requests it. These levels are not a measure of student ability; they are disclosure controls for a particular task.

Escalation requires a visible learner action or explicit request. The controller can maintain a per-task hint ceiling and may apply a cooldown after repeated requests if the course design requires it, but must not use a hidden psychological profile to restrict help. A learner can return to an earlier level or ask the instructor for direct assistance. If the evidence is ambiguous, the controller should lower certainty or request another tool run rather than escalating to a more specific repair. A confidently worded but unverified hint is not a reason to advance the ladder.

The sequence encourages productive effort while recognizing that an engineering lab may impose time and accessibility constraints. Course designers can configure a direct-explanation mode or an immediate worked example. A task policy can also require students to attempt a prediction before revealing a trace. These settings should be disclosed in the task page before work begins. The system records which configuration was active so that instructional evaluation does not treat differently scaffolded interactions as equivalent.

5.3 Hypothesis testing and feedback grounding

The feedback generator first classifies the immediate diagnostic and drafts a small number of candidate explanations. Each candidate identifies supporting evidence, disconfirming evidence when available, and a learner action that could distinguish it from alternatives. The selected ClaimSegment remains a hypothesis until the declared check supports it. If the evidence cannot distinguish causes, the system asks for a targeted inspection or instructor-authored test rather than presenting one explanation as fact.

The structural validator checks that each cited evidence identifier belongs to the current

© 2026 TRTEE. Open Access under CC BY 4.0

snapshot and run, that quoted output matches the stored excerpt hash, that approved-source identifiers are current, and that the proposed action is permitted by the task contract. It rejects absent files, stale records, hidden-test references, or course material outside the approved retrieval set. These are deterministic integrity checks. Whether an interpretation is semantically supported is assessed separately; an identifier's presence is not proof of entailment. Where possible, templates generate exact observations and the LLM supplies only a qualified explanatory bridge.

A useful feedback message is concise enough to act on yet complete enough to explain how the proposed action relates to the evidence. The interface uses three labeled parts: what the tool observed, one possible interpretation, and a learner-performable next step. A prediction question can encourage a testable model of execution. The article proposes this interaction structure but does not claim that it improves learning in a real course [30].

5.4 Feedback generation and validation procedure

The procedure uses the states in Table 5. Asynchronous queues must preserve the snapshot, run, evidence, response, and learner-action links. The baseline permits one initial model call and at most one shared repair call per request across all validators. A different course budget is a versioned protocol variant and must be evaluated for cost, latency, and accessibility; validator stages must not each reset the baseline allowance.

1. Bind the active task contract and policy version; freeze submitted files as an immutable snapshot.
2. In ToolsRunning, invoke only allowlisted adapters inside the declared resource and filesystem limits.
3. In EvidenceNormalized, store typed observations, source locators, hashes, and raw-output access controls.
4. Close if all required checks pass; otherwise bind the evidence packet and select the hint ceiling.
5. Ask the LLM for claim-level segments, a qualified hypothesis, and a learner-performable next action.
6. Check schema, reference existence and freshness, source integrity, disclosure, and task policy; do not treat linkage as semantic proof.
7. Recheck current request before display; exhausted repair budget → SafeFallback or InstructorEscalation.
8. Record the learner action; an edit starts a new run. ResolvedForNow records passed checks or the learner's explicit stop; stopping is not evidence that checks passed.

Table 5. Canonical feedback state transitions

Canonical state	Trigger and gate	Transition or failure path
Ready	Learner submits or requests a check; task	Valid contract → SnapshotFrozen; missing or invalid

© 2026 TRTEE. Open Access under CC BY 4.0

Canonical state	Trigger and gate	Transition or failure path
	contract and policy must exist.	policy → InstructorEscalation.
SnapshotFrozen	Create digest; bind task, policy, runtime, and file manifest.	Valid snapshot → ToolsRunning; digest or manifest failure → SafeFallback.
ToolsRunning	Broker invokes only approved adapters under fixed budgets.	Completed run → EvidenceNormalized; timeout or isolation failure → SafeFallback or InstructorEscalation.
EvidenceNormalized	Preserve raw output, typed diagnostics, source locators, and hashes.	All required checks pass → ResolvedForNow; otherwise usable evidence → HintDrafted; unusable evidence → SafeFallback or InstructorEscalation.
HintDrafted	LLM returns claim segments for this snapshot and run.	Schema valid → ResponseValidated; otherwise use the shared repair budget, then SafeFallback.
ResponseValidated	Check ID existence, freshness, source integrity, disclosure, and policy.	Checks pass → AwaitingLearner; failures use the same repair budget, then SafeFallback.
AwaitingLearner	Learner inspects evidence, requests help, edits, stops, or asks for the instructor.	Hint request → HintDrafted with current evidence; edit → RecheckRunning; handoff → InstructorEscalation; stop → ResolvedForNow.
RecheckRunning	Freeze a new snapshot and run the declared checks.	Completed run → EvidenceNormalized; timeout or isolation failure → SafeFallback or InstructorEscalation. Partial results are not passes.
SafeFallback	Model/tool timeout or invalid response; show only safe raw observations.	Offer retry or instructor route; never label an outage as success.
ResolvedForNow / InstructorEscalation	Close under the active course policy; retain the event trail.	Record checks_passed or learner_stopped separately; InstructorEscalation records handoff, not success.

5.5 Learner-facing response format

The feedback panel should make provenance legible without turning every response into a log dump. A compact header can identify the run and whether the result came from a compiler, test, analyzer, or debugger. The student can expand a detailed trace or test record, while the main message links to the relevant line, test name, or variable. Evidence details should be keyboard-accessible and compatible with screen readers. Color can reinforce status but must not be the sole carrier of meaning.

The message should distinguish observed output from a proposed explanation using explicit labels. For example: “Observed: E1 reports 0.0 °C at index 4; expected 25.0 °C. Possible explanation: the right boundary may have been assigned to the wrong index. Next step: inspect the assignment destination and predict the endpoint test result.” A separate control can ask for a stronger hint or send the episode to the instructor. The interface should avoid phrasing a model hypothesis as “the bug is” unless deterministic task logic establishes that conclusion.

© 2026 TRTEE. Open Access under CC BY 4.0

When the learner submits a change, the system summarizes what changed only if it can derive that summary reliably from a diff. It then reports the new run's outcome and asks the learner to connect the result to the original hypothesis. This makes the feedback loop visible and reinforces that the student—not the model—performed and verified the edit. A learner can mark the hint as confusing or incorrect; that signal is logged for review, but it is not used to penalize the student.

6. Synthetic Engineering Laboratory Example

6.1 Exercise specification

Consider a synthetic second-year engineering programming laboratory on steady one-dimensional heat conduction through a uniform rod. The rod is represented by five equally spaced nodes indexed 0 through 4. The left and right endpoint temperatures are fixed at 100 °C and 25 °C, respectively; there is no internal heat generation. For the discrete steady-state model, each interior node satisfies T_i = (T_(i-1) + T_(i+1)) / 2. The expected profile is therefore [100, 81.25, 62.5, 43.75, 25] °C. The task contract declares the units, node count, boundary conditions, permitted tools, endpoint check, profile tolerance of 0.01 °C, and a maximum of 500 iterations. This is a synthetic design fixture, not a real course or student submission.

The seeded defect assigns the right boundary to index n-2 instead of the final valid index n-1. With n = 5, this writes 25 °C to index 3 and leaves index 4 at 0 °C. The interior update loop is separately constrained to indices 1 through n-2 and preserves both endpoints. The test bundle checks array length, exact boundary values within a declared numeric tolerance, and the final profile. This isolates an indexing defect from a numerical-method failure.

The fixture defines a code snapshot, a task policy with hint ceiling 3, and a pinned Python environment. The initial boundary test runs before iteration and reports the final-node mismatch. A bounded trace records the boundary assignment destination and current snapshot. The LLM receives only the sanitized evidence packet and an approved explanation of fixed endpoint conditions. It cannot access hidden test source unless the instructor explicitly makes it visible.

© 2026 TRTEE. Open Access under CC BY 4.0

Seeded initialization code (the defect is marked):

```
def initialize_profile(n, left, right):
    T = [0.0] * n
    T[0] = left
    T[n - 2] = right  # seeded off-by-one defect
    return T
```

6.2 Evidence-to-feedback path

On the first run, evidence E1 reports expected 25.0 °C and observed 0.0 °C at index 4. After zero initialization, trace E2 records the right-boundary assignment to index 3, leaving index 4 at 0.0 °C. These observations do not prove the learner's intent, but support the hypothesis that the right boundary uses the wrong index. The response packet retains both evidence identifiers, the snapshot digest, the run identifier, and the active task-policy version.

At hint level 1, the system displays the mismatch and asks the learner to inspect the destination index. At level 2, it asks the learner to compare the valid index range 0…n-1 with the index used for the right boundary. At level 3, it asks the learner to write the corrected index in pseudocode and predict the endpoint test result. It does not insert a patch into the learner's file. Table 6 summarizes the evidence chain and decision points.

Suppose the learner changes the assignment to the final valid index, creates a new snapshot, and reruns the tests. The endpoint check then passes. The solver is run with the declared Jacobi update, keeping both endpoints fixed; the profile test checks the result against [100, 81.25, 62.5, 43.75, 25] °C within the stated tolerance. The system reports only those declared checks and asks the learner to explain why the endpoints are excluded from the update. It does not claim that these checks prove every implementation correct.

If the revised run still fails, the system compares the new run with prior evidence without reusing stale identifiers. If the trace shows the correct boundary index but the profile remains outside tolerance, it asks the learner to inspect the update rule or convergence result; if the evidence cannot distinguish causes, it abstains and routes the episode to the instructor. The fixture therefore tests both a supported hypothesis and a safe response when evidence is insufficient.

© 2026 TRTEE. Open Access under CC BY 4.0

Table 6. Synthetic evidence-to-feedback path

Step	Synthetic heat-conduction exercise
Task contract	Five equally spaced nodes; T0 = 100 °C and T4 = 25 °C; no internal heat generation; interior-node average update; tolerance 0.01 °C; maximum 500 iterations.
Seeded defect	Initialization assigns the right boundary to T[n-2] rather than T[n-1]; with n = 5, T[3] receives 25 °C and T[4] remains 0 °C.
Observed evidence	E1: index 4 is 0.0 °C; expected 25.0 °C. E2: the right-boundary assignment writes to index 3, leaving index 4 at its initialized 0.0 °C.
Qualified hypothesis	The right boundary may be assigned to the wrong array index; status remains unverified until a new snapshot is tested.
Hint path	L1: inspect the assignment target. L2: compare valid indexes 0…n-1 with the target. L3: write the corrected expression and predict the test result.
Learner revision	The learner changes the index, submits a new snapshot, and reruns the endpoint and profile checks; the assistant does not edit the file.
New evidence	Endpoint and profile checks pass within the declared tolerance; the record supports only these checks, not universal program correctness.
Reflection	The learner explains why endpoints remain fixed and proposes another check for the numerical profile.

6.3 What the example establishes and what it does not

The example establishes a concrete design trace for one synthetic boundary-index fixture: a task contract defines domain and test evidence; approved tools produce run-specific observations; the normalizer stores them; the model drafts claim-level segments; validators check traceability and policy; and the learner chooses and tests a change. It illustrates the difference between a passing declared check and general correctness. It is a reproducible design fixture, not evidence of model accuracy or learning benefit.

The example does not demonstrate that an LLM will diagnose every implementation correctly, that a hint is understandable to students, or that learners improve relative to ordinary feedback. No software deployment, participant data, or measured outcome is reported. Before a classroom pilot, a prototype should pass the hard integrity and policy gates in Table 7 and undergo separate semantic-quality, accessibility, privacy, and human-review checks.

7. Pedagogical Configuration for Engineering Laboratories

7.1 Task authoring and instructor controls

An instructor configures a task by declaring learning objectives, expected inputs, runtime,

© 2026 TRTEE. Open Access under CC BY 4.0

permitted tools, test meanings, likely diagnostic categories, hint levels, and escalation routes. The author should write at least one evidence-linked explanation for high-frequency failures and identify cases where the model must abstain. Examples include a test that can fail for multiple unrelated causes, code that relies on a domain assumption not represented in the prompt, or a numerical result that depends on floating-point tolerance. The author can also decide which tests are visible and when a worked example becomes available.

The configuration interface should let instructors preview each hint level against seeded code snapshots and tool outputs. A preview displays the evidence packet, model response, validator decisions, and final student-facing message. Instructors can edit task-specific language, mark a response unacceptable, or choose a fallback template. Versioned task policies make it possible to identify which model prompt and hint configuration generated a historical message. Approval should be required before a task is published to students.

A task can specify handoff conditions such as repeated failure after a bounded number of attempts, a resource-limit violation, unsupported tool output, ambiguous evidence, or a learner request. The handoff includes a concise evidence summary, exact snapshot and run identifiers, and prior learner actions so an instructor can reconstruct the episode. Instructors can correct a model-generated hypothesis, with the correction recorded as a separate event.

7.2 Learner control, accessibility, and transparency

Learners should be able to choose among predictable help actions: inspect tool output, request a small hint, request a stronger hint, explain their current hypothesis, rerun after editing, or contact the instructor. The interface should state when feedback is AI-assisted and what evidence the assistant can see. It should also state whether source files are retained, who can inspect them, and whether an external model service processes them. Transparency is a prerequisite for meaningful consent and trust calibration, although institutional policy determines the exact notice and consent model. Feedback uptake also depends on learners' ability to interpret and use comments, which is one reason the interface should make evidence, action, and learner choice explicit [29].

The design supports multiple modes of access. A learner can read or hear a concise diagnostic, navigate from a message to the referenced code region with a keyboard, and

© 2026 TRTEE. Open Access under CC BY 4.0

request an explanation that avoids color-only cues. A task may provide alternative ways to demonstrate the same debugging reasoning, such as annotating a trace instead of speaking aloud. The system should not infer an accommodation from interaction timing or require disclosure of a disability to unlock a different hint mode. Accessibility support is configured through the course and institution, with student-specific accommodation data kept out of the general model prompt. Feedback should also be concise and organized so learners can use it without unnecessary cognitive load; any such benefit must be verified for the task and learner context [31].

Feedback should be available in clear language while retaining exact technical terms where needed. A translation may help multilingual learners, but code identifiers, compiler messages, and unit symbols should remain linked to the original form. The learner can expand the raw diagnostic to compare it with the paraphrase. Where translation confidence or technical fidelity is uncertain, the system should preserve the original message and offer a human explanation rather than silently substituting a potentially misleading term.

7.3 Engineering reasoning and transfer

A programming laboratory can use tool-integrated feedback to make engineering reasoning visible, but the system should not reduce the objective to passing tests. A test may encode a narrow case, while engineering work often requires reasoning about assumptions, units, physical constraints, numerical stability, and model limits. Task authors can connect tests to those concepts and ask learners to articulate which assumptions the test exercises. The system can prompt that explanation after a run, but the student's response remains the primary evidence of reasoning and may be reviewed by an instructor [32].

Transfer requires learners to understand a principle beyond one code location. After resolving an indexing error, a learner might apply a similar boundary check to another array or explain how the same condition appears in a finite-difference discretization. The assistant can offer an instructor-authored variant after the original task is complete, but it should not automatically generate arbitrary new exercises whose correctness has not been reviewed. The course team can maintain a library of tested variants and attach each one to a specific objective. Problem-centered tasks and guided applications can connect such debugging work

© 2026 TRTEE. Open Access under CC BY 4.0

to broader engineering practice [33].

The proposed design is compatible with project-based, laboratory, and capstone settings, but the configuration burden differs. A short programming exercise can have deterministic tests and a narrow hint ladder. An open-ended design project may rely on simulations, domain-specific checks, or peer review and may not have one expected output. In such cases, the system should support evidence such as a bounded simulation trace or a declared constraint violation and should route judgment-heavy feedback to an instructor. The architecture does not assume every engineering activity can be fully automated.

8. Safety, Privacy, and Governance

8.1 Sandboxed code execution

Executing learner code creates a security boundary that cannot be delegated to a language model. The execution broker should run each attempt in a short-lived isolated environment with least privilege, a read-only base image, a task-specific writable directory, no network access by default, and strict CPU, memory, process-count, and wall-clock limits. It should prevent access to host files, credentials, container sockets, unrelated student data, and other tasks. Dependencies should be pinned and fetched during a controlled build rather than installed from an untrusted submission at run time.

These controls reduce exposure but do not guarantee containment. Container isolation depends on host configuration, kernel security, runtime patching, and operational monitoring. Institutions should assess the execution environment against their secure software development practices and threat model, including malicious code, denial-of-service attempts, dependency compromise, and escape attempts [34], [35]. A security incident path should stop execution, preserve relevant logs, notify the responsible operator, and avoid sending potentially sensitive artifacts to an external model provider.

Tool adapters must sanitize outputs before they reach the model or student interface. Stack traces and file paths can reveal infrastructure details; test fixtures may include information that should remain hidden. The broker should expose only task-approved spans and redact secrets or environment variables. The test runner must keep expected answers

© 2026 TRTEE. Open Access under CC BY 4.0

separate from the model context unless an instructor intentionally permits disclosure. Run output should be bounded to prevent an adversarial program from flooding storage or the language model with a huge response.

8.2 Prompt injection and untrusted content

Student code, comments, filenames, strings, and test output are untrusted data. Delimiting them in a prompt does not enforce a security boundary. The model has no tool credentials, cannot change policy, and cannot trigger arbitrary execution; the broker enforces its allowlist independently. Keep unauthorized test material out of prompts and fallback output. A model's disclosure label or a text filter cannot guarantee that free-form hints reveal no answer. Review actual content, use approved templates where strict disclosure is required, and test adversarial cases before release.

Retrieved documentation and error text can also contain malicious or misleading instructions. The retrieval collection should be curated and versioned, and retrieved passages should be quoted or delimited with source identifiers. A response can use a passage to explain a programming concept, but it cannot treat an instruction embedded in the passage as a change to system policy. Red-team tests should include direct and indirect prompt injection, hidden instructions in comments, malformed Unicode, and oversized outputs. Findings should be fixed in the task configuration, adapters, and model boundary rather than by asking students to avoid such inputs.

Abstention is required when evidence is contradictory, an approved source is unavailable, or a response fails validation. Present only authorized, sanitized observations, a safe next action, or an instructor route. If output cannot be safely exposed, provide a service-status notice instead. Neither the ordinary interface nor fallback may bypass access or disclosure rules. Distinguish availability from task progress: silence, an explicit stop, and an incomplete run provide no evidence that all checks passed.

8.3 Data minimization, retention, and accountability

The system should collect only data needed to provide and evaluate the feedback service. A pseudonymous learner identifier, task identifier, code snapshot digest, tool evidence, hint level, and learner action may be sufficient for the core workflow. Names, demographic

© 2026 TRTEE. Open Access under CC BY 4.0

characteristics, unrelated grades, and private course records are unnecessary for ordinary hint generation and should not be included by default. If a later evaluation needs additional data, it requires a separate governance process, approved research protocol where applicable, and clear notice to participants.

Retention periods should be explicit and proportionate. The institution can retain code and event records until the course appeal window closes, then delete or de-identify them according to policy. Instructors and support staff receive role-based access; model providers receive only approved prompt content under a reviewed data-processing arrangement. Logs used for debugging the service should avoid retaining full code longer than necessary. Students should be able to see what was stored and have a clear contact route for questions or correction requests.

Accountability requires a named owner for the course configuration, security controls, model integration, and incident response. A model card or system card can document intended use, limitations, evaluated failure modes, and model version. The institution should monitor changes in vendor terms and model behavior, and rerun regression tests before a model update is used for feedback. A risk-management framework can structure this work, but it does not substitute for institutional review, local privacy law, or professional judgment [36].

8.4 Academic integrity and fairness

A teaching assistant that provides code hints can affect how students complete graded work. Instructors should state which assistance is allowed, how AI support should be acknowledged, and whether the same interface is available to all students. The system should not secretly provide different hint ceilings based on inferred ability or risk scores. If course policy intentionally offers differentiated scaffolds, the basis and student-facing options should be transparent and reviewed for fairness. The system should not make misconduct determinations from similarity or interaction logs alone.

Model behavior can vary across programming languages, dialects, naming conventions, and the clarity of student explanations. Evaluation should therefore test representative code styles and accessibility modes, not only polished examples. Language models can reproduce

© 2026 TRTEE. Open Access under CC BY 4.0

biases or produce unreliable explanations; a system-level risk assessment should include harmful or dismissive feedback, misclassification of valid approaches, and uneven support. Instructors must have a way to inspect and correct such behavior, and students need an appeal path when an automated response materially affects their work.

The architecture separates formative assistance from grading to reduce the risk that an uncertain model output becomes a consequential decision. If an institution later connects the system to grades, that use requires additional validation, disclosure, human review, and an appeal process. A response that says “the model suggests this line may be the source” should not be converted into a grade or misconduct record. The system should retain the distinction between help given and evidence of achievement.

9. Implementation and Verification Plan

9.1 Prototype sequence

A first prototype should implement one programming language, one task family, and a small approved set of tools. The initial build can use deterministic templates for compiler and test feedback, then add an LLM only for converting structured evidence into alternative explanations and questions. This ordering permits a direct comparison between the baseline and model-assisted response without making the model responsible for execution or correctness. A prototype should use synthetic and instructor-authored submissions before any student code is processed.

The second stage can add debugger traces, a course-note retrieval store, and the hint controller. Each extension should have a versioned schema and regression suite. The third stage can integrate with a learning-management system, providing an assignment link, secure launch, learner consent notice, and instructor dashboard. Integration should not make the LMS responsible for executing arbitrary code; it should pass an authenticated task token to the isolated execution service and receive a bounded status response.

Before a pilot, the course team should conduct a threat model, privacy review, accessibility review, and instructional walkthrough. The environment should be load-tested against expected concurrent submissions and failure conditions. The course should prepare a

© 2026 TRTEE. Open Access under CC BY 4.0

fallback path for tool outages, including raw compiler output and instructor contact. A pilot should begin with non-graded practice or a clearly bounded formative activity, and only proceed to higher-stakes use after evidence supports the additional risk.

9.2 Functional, grounding, and security tests

Functional tests cover snapshot immutability, tool allowlists, timeout handling, output limits, normalization, and state transitions. Fixtures should include file mutation, same-snapshot reruns, policy changes during generation, superseding hint requests, duplicate events, late callbacks after stop or handoff, and exhausted shared retry budgets. They must reject stale publication and prevent partial runs from being counted as passes. Structural tests check IDs, source hashes, and declared limits. Test actual content for disclosure violations and semantic contradictions separately; valid labels do not establish safe or supported text.

Claim-support tests use seeded code snapshots and expected diagnostic facts. Reviewers see the task contract, original output, generated segments, evidence links, and policy. They label evidence-link validity separately from semantic support, hypothesis calibration, and action usefulness. The fixture set includes ambiguous failures, multiple defects, passing tests with an incorrect algorithm, malformed output, noisy warnings, and stale evidence. Report denominators and reviewer disagreement; do not collapse structural validity and semantic entailment into one score.

Security tests should include malicious submissions that attempt network access, process spawning, filesystem traversal, resource exhaustion, and environment inspection, as well as prompt-injection strings in comments and output. The model interface should be tested with attempts to reveal hidden tests or override the hint policy. Failures should be recorded separately for the sandbox, model, and validator so that a passing response check cannot obscure an execution isolation failure. Security testing should be repeated after changes to the runtime image, broker, tool adapters, or model provider.

9.3 Pedagogical and human evaluation

Once the software passes technical gates, educators can review response quality against a rubric covering evidence fidelity, diagnostic usefulness, clarity, hint level, domain

© 2026 TRTEE. Open Access under CC BY 4.0

appropriateness, accessibility, and preservation of learner agency. Reviewers should see the original tool output and task policy, not only the generated message. They can rate a response as acceptable, needs revision, or unsafe, and explain which evidence or policy was missing. Inter-rater disagreement should be retained and adjudicated; a single average score can conceal a critical subgroup of failure cases.

A subsequent classroom study would ask different questions from software verification. It might compare an LLM-assisted evidence interface with conventional tool output under equivalent task conditions, measuring process behavior, correctness on new tasks, quality of explanations, help-seeking, and perceived workload. A sound design would define the population, assignment conditions, learning outcomes, instructor role, consent, and analysis plan in advance. It would account for novelty effects, prior programming experience, task difficulty, and contamination between groups. The present article provides no such data and does not imply that the architecture improves learning.

The evaluation should examine whether students can distinguish observations from hypotheses and whether they can explain a repair after the assistant has helped. Process measures may include the order of tool runs, hint escalation, time between feedback and revision, and instructor handoff. These measures are not direct evidence of learning and should not be used as proxies without validation. Interviews or think-aloud protocols may help reveal whether students understand the evidence path, but they require appropriate ethical review and careful interpretation.

Cost and workload belong in the evaluation as well. Model latency, tool runtime, API cost, instructor review time, fallback frequency, and support requests can determine whether a design is operationally sustainable. An LLM may reduce the effort of rephrasing a diagnostic but increase the work of checking inaccurate suggestions. The system should report those trade-offs rather than counting generated messages as a productivity gain.

9.4 Prospective acceptance criteria

Before any pilot, the institution should define release gates. The proposed hard gates are: every seeded snapshot-mismatch, stale-reference, unauthorized-tool, hidden-test-disclosure, and hint-ceiling challenge is blocked; no response is displayed for a snapshot other than the

© 2026 TRTEE. Open Access under CC BY 4.0

one that generated its evidence; and every tested malformed-response or service-outage case reaches a safe fallback or instructor route. Any hard-gate failure blocks release. A separate predeclared threshold is required for human-rated semantic support and action quality; the threshold, sample, rubric, and adjudication process must be documented before testing. No result is claimed here. Table 7 lists the release gates and evidence to collect.

Pedagogical release criteria include instructor acceptance of seeded examples, accessible interaction review, understandable notices, and a published hint policy. The course team should document the service fallback, learner complaint route, retention schedule, access roles, and response to harmful feedback. Passing unit tests alone does not establish classroom readiness; operational support and human oversight are also required.

Table 7 combines deterministic checks with content-review gates. Passing seeded cases does not prove general disclosure safety or semantic correctness. These are prospective criteria; no implementation has been tested against them.

Table 7. Proposed prototype release gates and evidence

Gate	Minimum rule before pilot	Evidence to retain
Snapshot and evidence integrity	Every seeded stale-response, altered-excerpt, and cross-snapshot fixture is rejected; no response is shown for a different snapshot from its evidence.	Fixture results, snapshot/run identifiers, and failure logs.
Tool authority and resource limits	Every unauthorized adapter or prohibited operation in the test suite is blocked; timeout and budget failures do not continue silently.	Allowlist, timeout, filesystem, network, and resource-limit test records.
Disclosure and policy	Every seeded hidden-test disclosure and above-ceiling response is blocked; any hard-policy failure blocks release.	Policy-violation fixtures and validator decisions.
Safe fallback	Malformed output, model outage, tool outage, or unparseable evidence leads to a neutral observation, retry option, or instructor route; none is labeled success.	State-transition, fallback, and handoff records.
Semantic feedback quality	A course team sets its acceptance threshold before evaluation; claim support, uncertainty calibration, and action usefulness are rated separately with disagreement documented.	Rubric, sample description, reviewer ratings, adjudication, and uncertainty intervals.
Governance and accessibility	Privacy, retention, access, accessibility, notices, and complaint routes are approved before student code is processed.	Threat-model, access-control, deletion, accessibility, and course-policy reviews.

10. Discussion and Conclusion

10.1 Design implications

The central design choice is to make the language model a constrained interpreter of

© 2026 TRTEE. Open Access under CC BY 4.0

evidence rather than an unbounded code agent. Tool adapters provide reproducible observations; a normalizer gives those observations stable identifiers; the model drafts a hypothesis and an action; and a validator checks their links and allowed disclosure. This division makes the system easier to inspect and gives instructors a concrete surface for testing. It also makes failure modes local: a malformed trace, an unsupported explanation, and a policy breach can be diagnosed as different classes of problem.

A second implication is that feedback quality depends on the assignment contract. Without explicit expected behavior, test semantics, relevant concepts, and hint rules, the model has little basis for a trustworthy explanation. Authoring that contract requires faculty work and may be substantial for open-ended engineering projects. A system that hides this work behind a chat interface risks presenting an unreviewed generic assistant as if it understands the discipline. The proposed architecture keeps task authoring and approval visible as first-class functions.

A third implication is that pedagogical sequencing can be encoded without assuming one universal level of support. A staged hint ladder offers a default structure, while course policy can make direct explanation available. The system records what support was provided and invites learner action, but it does not infer that fewer hints always mean more learning. The relationship between hint level, performance, and learner understanding must be studied in context, not assumed from interface events.

10.2 Limitations

This manuscript is a design specification and has not been implemented or evaluated. Exact tool adapters, host-isolation mechanisms, model provider, identity integration, and deployment-specific thresholds remain to be selected. The claim-level schema and state transitions reduce ambiguity but do not establish semantic correctness; that must be assessed in a prototype using seeded cases and human review. The synthetic example is limited to one simple numerical boundary condition and cannot represent all language, simulation, hardware, or open-ended design tasks.

The design also assumes that course teams can author meaningful tests and explain their semantics. In open-ended design tasks, expected outputs may be partial or contested, and a

© 2026 TRTEE. Open Access under CC BY 4.0

fixed test suite can privilege one implementation style. Debugger traces can be large and difficult to summarize; a normalization error could omit the very evidence needed to diagnose a problem. LLM grounding checks can themselves be imperfect, so deterministic links and human review remain necessary. Sandboxing mitigates but does not eliminate execution risk.

Finally, the effectiveness of staged hints depends on course goals, student needs, assessment stakes, and instructor capacity. A hint that is productive for one learner may be inaccessible or insufficient for another. The design provides configuration and transparency but does not prove that those controls will be used consistently or fairly. Future implementations should test usability and safety before studying learning outcomes and should publish failures as well as successful interactions.

10.3 Conclusion

This article specifies a tool-integrated LLM architecture for stepwise debugging and feedback in engineering programming laboratories. It binds code snapshots to bounded tool runs, stores normalized observations in an append-only evidence/event record, and represents learner-facing observations, hypotheses, and actions as separate claim segments. Structural validators check evidence references, version freshness, source integrity, and course-policy limits; they do not claim to prove semantic entailment. Learners remain responsible for changing code, rerunning it, and explaining what the new evidence shows.

The proposal contributes a component architecture, a claim-level evidence schema, a canonical state protocol, a reproducible synthetic engineering fixture, a requirement traceability matrix, and prospective release gates. It does not report deployment, learning effects, or measured system performance. A prototype must first establish reliable tool execution, evidence fidelity, secure isolation, accessible interaction, semantic feedback quality, and acceptable instructor oversight. Only then can appropriately designed studies determine whether the approach improves debugging practice or engineering learning.

© 2026 TRTEE. Open Access under CC BY 4.0

Acknowledgements

Not applicable.

Author Contributions

Z. L.: Conceptualization, Methodology, Investigation, and Writing – Original Draft. Y. F. X.: Formal Analysis, Validation, Visualization, and Writing – Review & Editing.

Conflicts of Interest

The authors declare no conflicts of interest.

Generative AI Statement

Generative artificial intelligence was used to assist with language editing and drafting.

© 2026 TRTEE. Open Access under CC BY 4.0

References

[1] Ala-Mutka, K. (2005). A survey of automated assessment approaches for programming assignments. Computer Science Education, 15(2), 83–102. https://doi.org/10.1080/08993400500150747

[2] Keuning, H., Jeuring, J., & Heeren, B. (2019). A systematic literature review of automated feedback generation for programming exercises. ACM Transactions on Computing Education, 19(1), Article 3. https://doi.org/10.1145/3231711

[3] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. arXiv. https://doi.org/10.48550/arXiv.2107.03374

[4] Chen, X., Lin, M., Schärli, N., & Zhou, D. (2024). Teaching large language models to self-debug. In Proceedings of the International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2304.05128

[5] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In Proceedings of the International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2310.06770

[6] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., et al. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092–1097. https://doi.org/10.1126/science.abq1158

[7] Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. Computer Science Education, 13(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

[8] Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. ACM Transactions on Computing Education, 18(1), Article 1. https://doi.org/10.1145/3077618

[9] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. ACM SIGCSE Bulletin, 36(4), 119–150. https://doi.org/10.1145/1041624.1041673

[10] Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the SOLO taxonomy. In Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (pp. 118–122). https://doi.org/10.1145/1140124.1140157

[11] Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In Proceedings of the Second International Workshop on Computing Education Research (pp. 73–84). https://doi.org/10.1145/1151588.1151600

[12] Luxton-Reilly, A., McMillan, E., Stevenson, E., Tempero, E., & Denny, P. (2018). Ladebug: An online tool to help novice programmers improve their debugging skills. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (pp. 159–164). https://doi.org/10.1145/3197091.3197098

© 2026 TRTEE. Open Access under CC BY 4.0

[13] Ko, A. J., & Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In Proceedings of the 30th International Conference on Software Engineering (pp. 301–310). https://doi.org/10.1145/1368088.1368130

[14] Hattie, J., & Timperley, H. (2007). The power of feedback. Review of Educational Research, 77(1), 81–112. https://doi.org/10.3102/003465430298487

[15] Nicol, D. J., & Macfarlane-Dick, D. (2006). Formative assessment and self-regulated learning: A model and seven principles of good feedback practice. Studies in Higher Education, 31(2), 199–218. https://doi.org/10.1080/03075070600572090

[16] Shute, V. J. (2008). Focus on formative feedback. Review of Educational Research, 78(1), 153–189. https://doi.org/10.3102/0034654307313795

[17] Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. Journal of Child Psychology and Psychiatry, 17(2), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x

[18] VanLehn, K. (2011). The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. Educational Psychologist, 46(4), 197–221. https://doi.org/10.1080/00461520.2011.611369

[19] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. In Advances in Neural Information Processing Systems, 36. https://doi.org/10.48550/arXiv.2302.04761

[20] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. In International Conference on Learning Representations. https://doi.org/10.48550/arXiv.2210.03629

[21] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. (2023). Self-refine: Iterative refinement with self-feedback. In Advances in Neural Information Processing Systems, 36. https://doi.org/10.48550/arXiv.2303.17651

[22] Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 15–26). https://doi.org/10.1145/2499370.2462195

[23] Denny, P., Kumar, V., & Giacaman, N. (2023). Conversing with Copilot: Exploring prompt engineering for solving CS1 problems using natural language. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education (pp. 1136–1142). https://doi.org/10.1145/3545945.3569823

[24] Kazemitabaar, M., Chow, J., Ma, C., Ericson, B. J., Weintrop, D., & Grossman, T. (2023). Studying the effect of AI code generators on supporting novice learners in introductory programming. In Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, Article 455. https://doi.org/10.1145/3544548.3580919

© 2026 TRTEE. Open Access under CC BY 4.0

[25] Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2022). Automatic generation of programming exercises and code explanations using large language models. In Proceedings of the 2022 ACM Conference on International Computing Education Research (pp. 27–43). https://doi.org/10.1145/3501385.3543957

[26] Koutcheme, C., Dainese, N., & Hellas, A. (2025). Direct repair optimization: Training small language models for educational program repair improves feedback. Proceedings of the 20th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2025), 564–581. https://doi.org/10.18653/v1/2025.bea-1.41

[27] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems, 33, 9459–9474. https://doi.org/10.48550/arXiv.2005.11401

[28] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., & Fung, P. (2023). Survey of hallucination in natural language generation. ACM Computing Surveys, 55(12), Article 248. https://doi.org/10.1145/3571730

[29] Carless, D., & Boud, D. (2018). The development of student feedback literacy: Enabling uptake of feedback. Assessment & Evaluation in Higher Education, 43(8), 1315–1325. https://doi.org/10.1080/02602938.2018.1463354

[30] Chi, M. T. H., de Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. Cognitive Science, 18(3), 439–477. https://doi.org/10.1207/s15516709cog1803_3

[31] Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. Cognitive Science, 12(2), 257–285. https://doi.org/10.1207/s15516709cog1202_4

[32] Prince, M. J., & Felder, R. M. (2006). Inductive teaching and learning methods: Definitions, comparisons, and research bases. Journal of Engineering Education, 95(2), 123–138. https://doi.org/10.1002/j.2168-9830.2006.tb00884.x

[33] Merrill, M. D. (2002). First principles of instruction. Educational Technology Research and Development, 50(3), 43–59. https://doi.org/10.1007/BF02505024

[34] National Institute of Standards and Technology. (2022). Secure software development framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST SP 800-218). https://doi.org/10.6028/NIST.SP.800-218

[35] National Institute of Standards and Technology. (2024). Secure software development practices for generative AI and dual-use foundation models: An SSDF community profile (NIST SP 800-218A). https://doi.org/10.6028/NIST.SP.800-218A

[36] National Institute of Standards and Technology. (2023). Artificial intelligence risk management framework (AI RMF 1.0) (NIST AI 100-1). https://doi.org/10.6028/NIST.AI.100-1

© 2026 TRTEE. Open Access under CC BY 4.0