Curated Semantic Mutants: Multi-purpose Artifacts for Grading and Hinting Student Test Suites

Rebecca Williams Earle
Northeastern University
Oakland, USA
williams.rebe@northeastern.edu

Jonathan Bell
Northeastern University
Boston, USA
j.bell@northeastern.edu

Abstract

Mutation testing can evaluate a student's test suite, but traditional mutation tools do not encode which generated faults a course rubric should weight or what feedback should be delivered to students that do not detect a mutant. We report a human–LLM workflow that pairs each curated semantic mutant with an instructor-approved seed phrase for an on-demand LLM expansion. Deployed across two semesters of CS 3100, an introductory software-engineering course (N = 1,893 submissions), the workflow provided automated per-submission feedback about undetected curated faults. In a deployment with on-demand hint expansions, 46% of students requested at least one, at roughly three cents each. At the failing submission–unit-pair level, requested expansions were associated with improvement on the next submission at 1.62 times the rate of pairs with no request, though because requests were self-selected this is not a causal estimate. A same-code comparison also shows that the curated semantic-mutant set contains a fraction of the mutants a traditional mutation engine produces. We do not compare this LLM-assisted approach with an instructor curating and writing mutants and hints by hand; whether it is cheaper or better remains an open question.

CCS Concepts: • **Social and professional topics** → **Computing education**; • **Software and its engineering** → *Software testing and debugging*.

Keywords: computing education, mutation testing, LLM, automated feedback, formative assessment, autograding

ACM Reference Format:
Rebecca Williams Earle and Jonathan Bell. 2026. Curated Semantic Mutants: Multi-purpose Artifacts for Grading and Hinting Student Test Suites. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3841644.3842688

SPLASH-E '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2949-2/2026/10
https://doi.org/10.1145/3841644.3842688

1 Introduction

Mutation testing changes a reference implementation and measures how many changes a test suite detects as faults [1]. PIT is one engine that implements this technique [2]. Large-scale evidence links mutant detection to real-fault detection, with limits [12]. In project-based courses, mutation testing helps distinguish fault-finding tests from happy-path coverage. Perretta et al. [18] evaluated it on 2,711 student submissions in C++, JavaScript, and TypeScript and found that mutation score is a strong proxy for the detection of instructor-seeded faults and a moderate proxy for the detection of faulty student implementations. Earlier deployments reported comparable utility [26, 30].

The faults these engines insert are single-line edits a syntactic operator can produce, such as a flipped comparison, a swapped constant, or a removed call. On the four assignments we study, default PIT [2] generates 152 to 236 mutants per assignment when targeting the graded classes. One class alone receives 95. The corresponding curated semantic-mutant sets contain 21 to 36 mutants (Table 2). For course grading, an instructor must still decide which generated faults represent specification contracts, how they contribute to the rubric, and what students should learn from surviving them. Classroom studies have evaluated curated mutation. Hall and Baniassad [7] graded student test suites with a small set of hand-written mutants and reported that mutation-graded students wrote more correct tests and engaged more closely with the specification. Prasad et al. [21] found that mutants useful for students encoded common *misconceptions*, deliberately "not those produced by mutation-testing tools."

Turning a surviving mutant into actionable feedback is a separate problem. Perretta et al. [19] deployed an in-platform system that showed students instructor-written hints about the mutants their tests failed to detect, across 4,122 students, and measured a small but statistically significant gain in mutant detection. That system asked instructors to hand-author a sequence of hints of increasing specificity for each mutant. This approach keeps instructor effort constant in the number of students, but the instructor must repeat the work for each assignment: choose the mutants to grade and write the hint sequence for each one.

Both tasks require an instructor to identify a meaningful semantic fault. In our workflow, an LLM proposes short *seed phrases* that characterize plausible specification misreadings

and the test suite gaps they leave. The instructor reviews and edits those proposals. For each instructor-approved seed phrase, the LLM drafts a whole-class source variant that encodes the same semantic fault. After instructor review, that variant becomes a curated semantic mutant. That mutant grades the test suite. At feedback time, an LLM expands its paired seed phrase against the assignment specification.

We deployed the pipeline across Spring and Summer 2026 in CS 3100, an introductory, project-based software engineering course taught in Java at Northeastern University. The dataset contains 1,893 graded submissions with operator and curated semantic-mutant kill rates. We compare the mutant sets on the same code, identify curated semantic mutants that student tests miss, and describe use of the LLM expansions. Each semantic fault links a grading mutant to an approved hint seed. We report what building and running this workflow in a live course required and what it did and did not demonstrate. §3 presents the approach. §5 reports the evaluation.

Research Questions

RQ1 (Operator vs. curated mutants). When applied to the same graded classes and student test suites, how do operator mutants and the deployed curated semantic mutants compare in number and per-submission kill rate, and how often do student suites fail to reach operator-mutant locations?

RQ2 (Difficulty). Which curated semantic mutants do student test suites systematically fail to detect, and what specification contracts do those misses share?

RQ3 (Deployment). How often do students request an LLM expansion of a mutant-grounded seed phrase, and how is a request associated with improvement on the next submission?

Contributions

1. **A reviewed curation workflow** in which an LLM proposes a semantic fault and drafts its implementation while an instructor reviews the proposal, validates the mutant, and approves its paired seed phrase. The mutant grades the test suite; the seed phrase seeds the hint.
2. **A same-code, cross-cohort comparison** of operator and curated semantic mutants. We report how many of each an assignment produces, how often student tests reach the operator mutants, and per-submission kill-rate associations.
3. **An experience report** deploying mutant-grounded hints in one cohort, reporting adoption, cost, and the observational association between a self-selected request and next-submission improvement.
4. **Two implementations**: PreBakedMutator, a PIT extension running whole-class source variants via ASM constant-pool rewriting, and a closed-input three-tier hint pipeline whose grading-action payload is reproducible from a pair of git SHAs and whose LLM call sees only the spec and seed phrase.

2 Background and Related Work

Mutation testing and the coupling effect. Mutation testing scores a test suite by the fraction of seeded faults (*mutants*) it detects [4, 10]. Engines such as PIT [2] and Major [11] produce *operator-generated mutants* through single-site bytecode rewrites, such as flipping a conditional or swapping a return. The score relies on the *coupling effect*: a test that detects simple faults also detects complex faults built from them [4]. A large-scale study found a correlation between mutant and real-fault detection, but some real faults coupled to no generated mutant [12]. Perretta et al. [18] reported a related classroom result across three institutions. Most student faults couple to operator-generated mutants. Reproducing the remaining faults requires an implementation structured unlike the reference.

Computing-education deployments have used operator-generated mutants to grade student suites [26, 30]. The Examplar line's *chaffs* establish the use of instructor-authored, whole-implementation faults [29], which resemble our curated semantic mutants. Two efforts focused on pedagogical curation. Hall and Baniassad [7] hand-wrote a small set of mutants to grade student-written test suites. Prasad et al. [21] instead derived *conceptual mutants* from clustered student misconceptions to diagnose what learners misunderstood. In each, a curated artifact serves a single purpose, either scoring a suite or diagnosing a misconception. LLMs have generated faults beyond those provided by operator engines [3, 27]. In the closest design to ours, Maton et al. [17] prompted an LLM to reimplement a method as a "programmer misunderstanding" and reached faults that operators missed. Their pipeline remained fully automated and targeted practitioner suites. Our workflow adds instructor review to LLM drafting of each whole-class source variant. Every approved semantic fault has two linked representations: a curated semantic mutant that grades a student's test suite and an instructor-approved seed phrase for the LLM expansion. This pairing ties each fault to a course specification.

LLM and instructor feedback in programming courses. LLM-driven hinting has been examined in next-step hint generators [16, 24] and within the programming-process telemetry tradition [9]. Generative AI has also entered computing courses [22], including project-based software-engineering classes such as ours [25]. Prior studies at this venue examined how autograder feedback changed students' responses to results [31] and whether synthesized feedback indicated that a student was on the right track [5]. These studies targeted implementation progress rather than the quality of students' own tests. CodeHelp [15] and CodeAid [13] withhold complete solutions while retaining the student's source

and a failing test in the model's context. Our system instead targets test suite quality. We give the model only the specification and an instructor-approved seed phrase, and withhold student code entirely. Our delivery architecture builds most directly on FeedBot [32], which established in-platform, formative, design-focused LLM feedback on programming assignments. We adopt that platform setting and ground each hint in a curated semantic mutant rather than the student's submission. The same mutant also grades the student's test suite. The closest precedent for instructor knowledge delivered at a failing test is Perretta et al. [19], whose system showed a hand-authored hint sequence for each undetected mutant. Our instructor-approved seed phrase plays the role of their first hint, and an LLM expands it against the specification. Every student in our deployment already saw the seed phrase at each failing test. Thus, we evaluate requests against that existing feedback rather than against no feedback. The course's prior pinned-thread help provides context, not a controlled comparison arm. The Design Recipe framing [6, 32], recently revisited with LLMs [14], informed the strategy comparison (§4) but was not the production prompt.

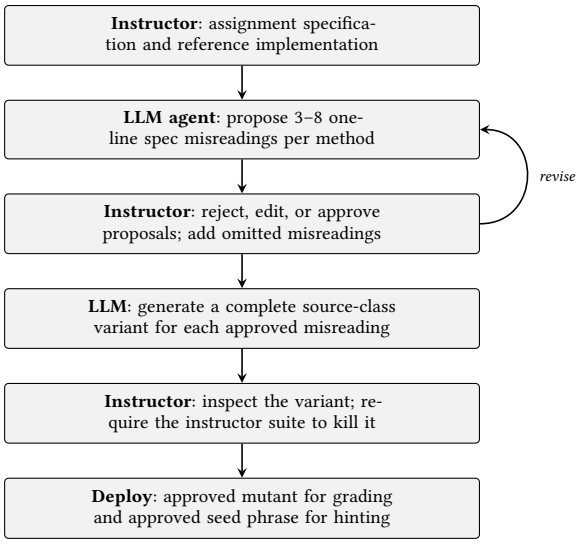


Figure 1. The mutant curation loop. The LLM proposes spec misreadings and generates source-class variants. The instructor reviews every phrase and variant, adds omitted misreadings, and applies the kill-gate before deployment.

3 Approach

3.1 Instructor–LLM Curation

We author semantic mutants through the loop in Figure 1. The instructor begins with the assignment specification and a reference implementation. Given the specification, an LLM agent (Claude Opus 4.6 via Claude Code in our deployment) proposes three to eight plausible misreadings per method as one-line hypotheses. One example is *"a student might think that imported recipes get a new ID assigned, rather than retaining their original ID."* The instructor rejects uninteresting proposals, edits promising ones, and adds misreadings that the agent omitted.

Each reviewed phrase becomes an instructor-approved seed phrase (§3.3). The LLM then receives the reference implementation and generates a whole-class source variant that encodes the approved misreading. The instructor inspects the variant and requires the instructor suite to kill it. This *kill-gate* confirms that the variant compiles and differs observably from the reference. The inspection checks that it implements the intended fault. A mutant killed by a test cannot be an *equivalent* mutant, which would behave like the reference on every input.

The "generate faults per method" instruction only scaffolds proposals. Some contracts concern a constructor or class; others span methods. We neither require a mutant for every method nor report a method-level average; assignment counts are the stable record.

In retrospect, the participating instructor estimates about one hour per assignment for reviewing proposals, validating mutants, and editing seed phrases. Section 6 discusses the limits of this estimate. The process produced 21 to 36 deployed mutants per assignment, or 117 across the four Spring assignments.

3.2 Curated Semantic Mutants and PreBakedMutator

The course is built around a recipe-management application, *CookYourBooks*, which students extend across four assignments (A1–A4). A1 builds the domain model of units, quantities, and ingredients; A2 adds unit conversion and scaling; A3 adds JSON-persisted recipe collections; and A4 adds a text parser and a service facade coordinating them. Each assignment has its own plausible misreadings, which the curated semantic mutants target. We measure how often students miss these mutants (§5.2) and compare their kill rates with operator-generated mutants (§5.1).

We use A2 as the running example, analyzed in depth in Section 5. Its specification includes several contracts that are easy to misread. For example: scaling a recipe by a factor multiplies its measured amounts (two cups of flour become four) but leaves *vague* amounts such as "a pinch of salt" untouched. A separate operation scales the whole recipe until one named ingredient reaches a target amount, converting between units where needed to compare them. Conversion runs through a registry of rules, some specific to a single ingredient (a cup of flour and a cup of sugar do not weigh the same) and each carrying a priority, so a house rule wins over a recipe rule and both over a standard one; when no rule applies, the conversion reports a failure rather than guessing.

Table 1 shows six representative mutants for A2. Each row lists the mutant code, the spec rule it violates, and the survival rate in students' graded submissions in Spring 2026 ($N = 435$ students submitted).

Table 1. Representative semantic mutants from A2 (Spring 2026; $N = 435$ submitted, of 408 enrolled at term end). Survival is the share of *active* submissions (each student's submission marked for grading, usually the latest) in which the mutant survived.

Code	Spec rule violated, and what the mutant does instead	Surv.
`SC1_ScalesVague Ingredients`	VagueIngredients must stay unchanged when scaling; mutant scales them as if measured.	7%
`SC2_No Instruction Scaling`	IngredientRefs in instructions must be scaled; mutant leaves them untouched.	7%
`ST3_WrongRule Priority`	Recipe rules register at RECIPE priority; mutant uses STANDARD.	48%
`CV2_NoIngredient NameInConvert`	Ingredient name must pass to the conversion registry; mutant omits it, so ingredient-specific rules never fire.	17%
`CG2_Matches Ingredient SpecificWithout Context`	Without an ingredient name only generic rules apply; mutant matches ingredient-specific rules in any context.	24%
`CI4_Wrong ExceptionType`	Conversion failure with a name throws the ingredient-specific exception; mutant throws the unit-only type.	33%

```
// Reference, in findAll:
- return objectMapper.readValue(
-     path.toFile(), RecipeCollection.class);

// JC1_CookbookTypeNotPreserved:
+ ObjectNode jsonNode = (ObjectNode)
+     objectMapper.readTree(path.toFile());
+ if ("cookbook".equals(
+     jsonNode.get("type").asText())) {
+   jsonNode.remove("type");
+ }
+ return objectMapper.readValue(
+     jsonNode.toString(), RecipeCollection.class);
```

Figure 2. Excerpt from the A3 whole-class mutant `JC1_CookbookTypeNotPreserved`. The mutant repeats the added block in both loading paths. The complete source variant is included in the supplement.

Each curated semantic mutant is a whole-class source variant, not necessarily one source-level edit. Figure 2 shows part of an actual A3 mutant. The reference implementation's recipe repository deserializes a saved collection directly. The mutant first removes the JSON type discriminator for a `Cookbook`, so loading it no longer preserves that subtype. The LLM made this change in both `findById` and `findAll`. These are multiple syntactic edit sites, but they encode one intended semantic fault: failure to preserve the `Cookbook` type. We do not classify such a variant as a higher-order mutant because it does not deliberately combine independently selected faults.

PIT [2] normally generates a mutant by applying a mutation operator to an existing class's bytecode [20]. We extended PIT with *PreBakedMutator*, an engine that swaps in whole-class source variants at runtime. For a source class `Foo`, an instructor can compile variants such as `Foo_WithSomeBug` as part of the normal build. At mutation time, PreBakedMutator patches a selected variant in place of `Foo`. This mechanism permits semantic faults that require more than a single operator edit. The extension adds one package of about 311 lines and touches six existing PIT files.

The one architectural obstacle is the JVM's class-identity constraint. PIT swaps each mutant into the test classloader with JVMTI `RedefineClasses`, which requires the replacement to share the name, supertype, and shape of the class it replaces. A variant of `Recipe`, for example, is compiled as a distinct class such as `Recipe_SC1` and has its own `this_class` entry. PreBakedMutator reads the variant's bytes and rewrites the constant pool through an ASM `ClassRemapper`. The remapper changes every internal reference back to the source class's identity. The JVM therefore sees the reference class's identity while it runs the variant's bytes.

The rewrite covers the whole constant pool rather than a single edit site. It handles supertype and descriptor references, lambda call sites, and nested or inner types recorded in the `InnerClasses` attribute. In principle, the same mechanism could support variants that coordinate edits across several classes, although our deployment did not require them.

3.3 Seed Phrases and LLM Expansions

The curated mutant and its seed phrase are declared together, in one file. Each assignment's solution repository carries an `autograder.yml` that the instructor edits directly (Figure 3). Its `gradedUnits` block groups mutants into *graded units*: the rubric line items a student sees, each with its own point value, scored by how many of the unit's mutants the student's suite detects. Its `mutantAdvice` block pairs each mutant with the instructor-approved seed phrase and names the whole-class variant PreBakedMutator swaps in. A mutant name appearing in both blocks is what makes the curated fault multi-purpose. The `locations` entry scores the test suite; the `prompt` entry seeds the hint.

For each graded unit with surviving mutants, the action joins the survivors against their seed phrases by target class and appends them to the test's error output, capped by `maxMutantHints`, a per-assignment setting: one phrase on A1, two on A2, and eight on A3 and A4. This output is student-visible, so every student whose submission leaves a mutant undetected sees the seed phrase inline in the test-failure feedback, whether or not they request an LLM expansion.

When a student requests an expansion, the action assembles the prompt from a fixed *template* with four parts: a role-and-rules preamble that does not vary between assignments, the assignment specification, one swappable strategy block, and the failure body carrying the inlined seed phrase. The preamble fixes the model's role and its disclosure, scope,

```
maxMutantHints: 2

feedbot:
  prompt: chain_of_thought   # default; left
      # unset in all four deployed assignments

gradedUnits:
  - name: "Test Recipe.scaleToTarget()"
    linearScoring: {points: 6, total_faults: 4}
    locations:
      - "Recipe  Recipe_ST3_WrongRulePriority"
      # + the unit's 3 other mutants

mutantAdvice:
  - name: "ST3_WrongRulePriority"
    prompt: "Your test suite did not detect
      that recipe-specific conversion rules
      are being added at the wrong priority
      level. Recipe rules should be at RECIPE
      priority, not STANDARD."
    sourceClass: "Recipe"
    targetClass: "Recipe_ST3_WrongRulePriority"
```

Figure 3. Excerpt from A2's `autograder.yml`, the file an instructor edits. The mutant `ST3_WrongRulePriority` appears in both blocks: once under `gradedUnits`, where it scores a rubric line item, and once under `mutantAdvice`, where its `prompt` is the seed phrase an LLM later expands. The `feedbot` block selects the strategy; every deployed assignment omitted it and so ran the default. The `app.cookyourbooks.model.` package prefix is elided throughout; the seed `prompt` is verbatim.

and tone rules; only the strategy block and failure body vary between calls. The rendered prompt, with the model, account, and rate-limit settings, is serialized into the autograder feedback JSON as a *prompt payload*. The hint server consumes the payload, dispatches the LLM call, persists the result, and renders the hint button; the LLM is never called inside the grading action.

In our deployment, the LLM does not receive the student's source code or tests. It also does not receive the reference implementation, the hidden instructor suite, or prior hints from the assignment. The instructor has reviewed and approved every seed phrase, so the expansion begins from an approved statement of expected behavior. The model uses the specification to elaborate that statement and can prioritize among gaps that co-occur on a submission.

Withholding code was a pedagogical choice rather than a technical requirement. It also eased coordination among the course's instructors, who agreed on this configuration more readily than one that included solution details. The closed input set keeps prompts small and avoids sending student code to the provider. However, there is no structural limitation preventing student or solution code from passing to the LLM.

The template keeps the strategy in its own slot because the formative study (§4) surfaced more than one workable formulation. Rather than fix the prompt on the one we deployed, the slot lets an instructor select among them per

```
<example label="acceptable">
"Review the spec's rules for how
decimal amounts should be simplified
in toString()."
</example>
<example label="unacceptable">
"The expected output is '1 cup', not
'1.0 cup'."
</example>
```

Figure 4. Two of the six acceptable/unacceptable disclosure contrasts in the preamble: the acceptable variant points at the spec section; the unacceptable one reveals the value the autograder expects.

assignment. Two strategies for prompting ship as built-in presets, and any other string the instructor supplies is used verbatim in their place. The *chain-of-thought* strategy, our default, asks the model to reason silently through five steps before writing: locate the failure, identify the relevant specification rule, decide whether the gap is in the test expectation or the implementation, name the underlying principle, and choose a single next action. The *checklist* strategy instead asks the model to pick one of four diagnostic angles—where the failure is, what the spec requires, what category of issue it is, or what condition causes the divergence—and write from that angle alone. A third strategy, a *Design Recipe* grounded in Zhu et al. [32], scored strong on specification alignment but weaker at giving a concrete next action (§4).

The instructor picks the strategy in the same file's `feedbot` block, naming one of the two presets or supplying a custom string that is inserted verbatim into the strategy slot while the rest of the template stays fixed. Per-assignment A/B work on prompt strategy is thus a configuration edit, not a fork of the pipeline.

The preamble's most important rule is an intended non-disclosure boundary with worked examples. We give the LLM six acceptable/unacceptable contrasts (Figure 4) that distinguish guidance toward the relevant specification rule from disclosure of the value the autograder expects.

The acceptable variant names the specification section without quoting its content. Figure 6 shows a residual risk: its seed names `RECIPE` and `STANDARD`, which the expansion repeats. A prompt cannot retract information already present in an approved seed. A separate rule restricts each hint to one issue, while the assignment's `maxMutantHints` setting caps how many seed phrases a submission shows. The tone rule forbids the LLM from naming the autograder, mutation tooling, or reference implementation; the mutant-grounded provenance is hidden from the student. The output is 3–4 sentences of plain prose ending with `Next step:` and one concrete action; a literal `RETRY` sentinel triggers server-side fallback.

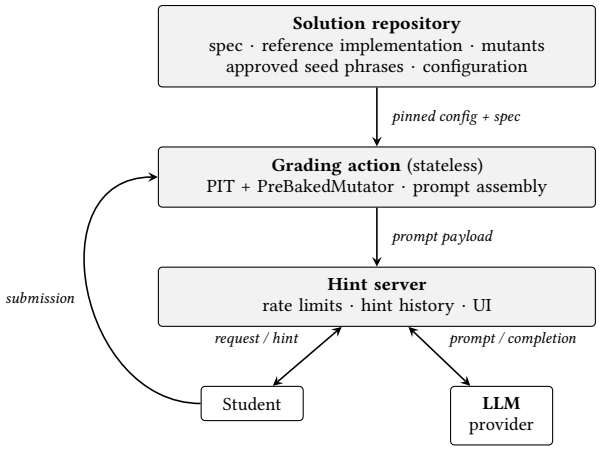


Figure 5. The integrated pipeline. The grading action receives versioned course artifacts and emits the prompt payload; the server handles provider calls and student interaction. The grading action never calls the LLM or holds provider credentials.

3.4 Integrated Deployment Architecture

The pipeline has three loosely coupled tiers (Figure 5). The solution repository holds the specification, reference implementation, semantic mutants, approved seed phrases, and per-assignment configuration. A stateless grading action runs PIT and assembles the prompt payload for each submission. The hint server dispatches the LLM call, enforces rate limits, persists the result, and renders the hint.

Each tier is independently versioned. The same student and solution git SHAs reproduce the same action payload, while the server keeps provider credentials out of the instructor's assignment repository. Instructors therefore do not manage API keys, and a provider failure does not prevent the action from grading the student's test suite.

4 Formative Prompt Development: Failure Modes and Guardrails

Before deployment, we used fourteen retained Spring 2026 student errors to develop the prompt and identify ways an expansion could fail. We generated 168 hints by crossing the errors with three exploratory strategy blocks and four candidate models (Claude 3 Haiku, GPT-4o mini, DeepSeek Chat, and Gemini 2.5 Flash Lite). The strategies explored different ways to organize the same specification and seed phrase. The chain-of-thought block used silent intermediate steps. The checklist constrained the model to one diagnostic angle. The Design Recipe variant translated the pedagogical structure of the platform's prior feedback approach into a prompt block [32]. They were engineering alternatives, not an exhaustive set of instructional strategies. A single reviewer scored the corpus using a rubric—Spec Alignment, Diagnostic Accuracy, Student Agency, Action Clarity, Constraint Compliance, and Clarity & Tone—developed iteratively during scoring.

The review exposed five recurring failure modes. *Blame-the-reference* directed a student to distrust the reference implementation or grading system rather than examine the relevant contract. *Spec contradiction* advised behavior inconsistent with the assignment specification. *Over-prescription* revealed an expected value or dictated so much of a test that little reasoning remained for the student. *Format drift* ignored constraints such as the three-to-four-sentence limit or the required final next step. *Vague guidance* repeated the failure without identifying a concrete testing action.

Section 3.3 describes the guardrails these observations motivated. Grounding the call in the specification and an approved seed phrase addresses unsupported diagnoses and spec contradictions. The prohibition on naming the reference or autograder addresses blame-the-reference. Worked contrasts and the single-issue rule limit over-prescription. The length, `Next step:`, and `RETRY` requirements address format drift and vague guidance. These are prompt controls, not guarantees. A model may still misread the specification, disclose too much, or ignore an output constraint. The formative study therefore supplies a failure taxonomy and design rationale, not evidence that the guardrails eliminate those failures.

The Summer 2026 deployment used the chain-of-thought block with Claude Sonnet 4.6, which was not among the four candidate models. We did not rubric-score the deployed corpus, so the formative results neither compare the production model with the candidates nor evaluate the quality of the hints students received. The single reviewer, iteratively developed rubric, and lack of a production-corpus review further limit the study; the full rubric, outputs, and failure-mode taxonomy appear in the supplemental material.

Figure 6 traces one mutant end to end, from the spec rule it violates through the instructor-approved seed phrase to the hint a student received in deployment.

5 Evaluation

RQ1 compares operator and curated mutants across both cohorts (§5.1). RQ2 describes difficulty among the curated mutants (§5.2). Both analyses use each student's active submission per assignment. RQ3 analyzes submission–graded-unit pairs across each student's trajectory in the hint deployment (§5.3).

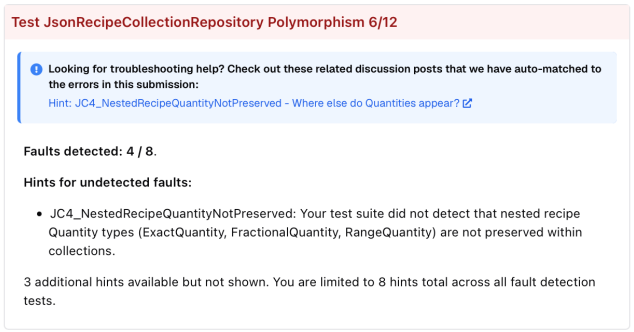
The course's prior feedback channel was an instructor-pinned discussion thread. In Spring, instructors pinned discussion threads to errors that recurred in office hours, and Pawtograder, the course's grading platform, linked a matching thread from the failing test (Figure 7). Instructors authored threads on 15 graded units across the four assignments (A1–A4). Nothing constrained what an instructor posted, but threads generally took one of two forms. Some

Spec rule (A2): recipe-specific conversion rules are registered at `RECIPE` priority.

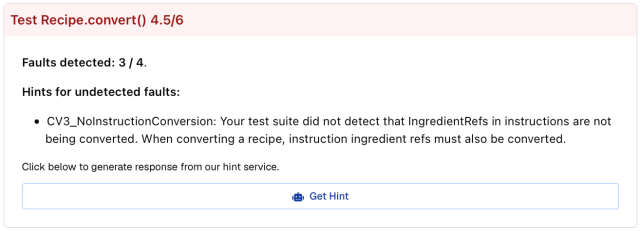
Seed phrase (mutant `ST3_WrongRulePriority`, verbatim from `mutantAdvice`): *"Your test suite did not detect that recipe-specific conversion rules are being added at the wrong priority level. Recipe rules should be at RECIPE priority, not STANDARD."*

Deployed hint (Summer 2026, verbatim, lightly elided): *"Your test suite is missing coverage for a specific contract defined in the spec around how conversion rules are applied during scaleToTarget. … Re-read the Recipe Transformations section, specifically the requirement about which priority level recipe-specific rules must use… Next step: Write a test for scaleToTarget that adds a conversion rule at RECIPE priority and verifies that the scaling result depends on it—for example by also registering a conflicting rule at STANDARD priority and confirming the RECIPE-level rule takes precedence."*

Figure 6. From curated mutant to deployed hint for `ST3_WrongRulePriority`, the hardest-surviving mutant in both cohorts (Table 1). The LLM saw the specification, the failing unit's output with the seed phrase inlined, and nothing else.



(a) Spring 2026: the inline seed phrase, plus an auto-matched pinned discussion-thread link (top, blue).



(b) Summer 2026: the same seed phrase, with a Get Hint button that expands it via the LLM on request.

Figure 7. What a student saw on a failing graded test in each term. Both screens show the instructor-approved seed phrase. The additional displayed resource is a pinned thread (a) or an on-demand LLM hint (b). The cohorts differed in other ways, so the panels are not a controlled comparison.

diagnosed why a test errored rather than failed. For example, one explained why a serialization test that passed against a student's own class threw on an extra field in the reference implementation's JSON. This post also gave the `ObjectMapper` setting and `@BeforeEach` placement that would fix the common error. Others expanded one mutant by hand. On A3, after many students detected seven of the polymorphism unit's eight mutants and missed `JC4_NestedRecipeQuantityNotPreserved`, an instructor quoted that mutant's seed phrase and expanded it. The thread asked where else in the domain model a `Quantity` appears and pointed at `Recipe`'s fields. It then posed the round-trip question a test would have to ask. Such a thread is an instructor-written expansion of the same seed phrase the LLM receives, written once for the cohort and only after the miss was visible. We describe the channel here for context and report its uptake in §5.3, without treating it as a comparison condition for the LLM expansions.

We deployed the complete pipeline in CS 3100 in Summer 1 2026, a six-week intensive term (N = 109 enrolled; cf. Spring 2026 N = 408; submitter counts below exceed enrollment because they include students who later dropped). The seed-phrase layer was identical in both cohorts: On every assignment, a student whose submission left a mutant undetected saw the instructor-approved seed phrase inline at the failing test, under the same per-submission `maxMutantHints` cap. In Summer 2026 students could additionally request an LLM expansion of the seed phrase, deployed on A2 only (Figure 7). This request was optional and students self-selected. The study was approved by our institution's IRB.

5.1 RQ1: Comparing Operator and Curated Mutants

For every active submission (the one marked for grading, by default the student's latest), we paired the student's tests with the canonical source used by the grader. Then, we ran both the default PIT mutation engine (the `gregor` engine with the `STRONGER` mutator group) and the curated prebake engine on the same four to six graded classes, excluding the surrounding handout and library code. We excluded submissions whose tests did not compile against the canonical source, leaving N = 1,579 Spring 2026 submissions (A1–A4) and N = 314 Summer 2026 submissions (A2–A4).

Table 2 reports the number of mutants generated by each engine for each assignment and their kill rates. The clearest difference is how many mutants each engine produces. In Spring, one default PIT run generated 152 to 236 mutants per assignment, compared with 21 to 36 curated mutants; A4's `IngredientParser` alone received 95 operator mutants. Operator mutants also frequently occurred at locations a particular student's tests did not execute. For each operator mutant, we computed the fraction of cleanly graded Spring submissions for which PIT reported `NO_COVERAGE`, then averaged those fractions without weighting across mutants. The result was 47% for A1, 39% for A2, 24% for A3, and 22% for A4.

Table 2 also reports descriptive correlations between the two kill rates across submissions. We report them for completeness and draw no conclusion from them. They range

Table 2. Per-submission operator-vs-curated kill-rate correlations, Spring 2026 (sp26) and Summer 2026 (su26). N is the number of cleanly-graded submissions; M_{op} and M_{cur} are the number of operator and curated mutants generated on the assignment's graded classes; r is Pearson correlation with a 95% Fisher-z confidence interval and ρ is Spearman correlation; $\overline{\text{op}}$ and $\overline{\text{cur}}$ are mean active kill rates for operator and curated mutants. A1 was not offered in su26. su26's A4 folds text parsing into the `RecipeService` facade rather than the separate parser classes sp26 graded, so it mutates fewer classes; its operator kill rate and correlation are not comparable across cohorts and are omitted ($M_{\text{op}} = 45$).

Asg	Cohort	N	M_{op}	M_{cur}	r [95% CI]	ρ	$\overline{\text{op}}$	$\overline{\text{cur}}$
A1	sp26	387	174	28	0.62 [0.56, 0.68]	0.27	0.81	0.97
A2	sp26	410	152	21	0.85 [0.82, 0.88]	0.56	0.71	0.81
A2	su26	107	152	21	0.84 [0.78, 0.89]	0.54	0.73	0.83
A3	sp26	386	236	32	0.54 [0.47, 0.61]	0.27	0.78	0.95
A3	su26	111	225	23	0.74 [0.64, 0.81]	0.51	0.73	0.92
A4	sp26	396	206	36	0.28 [0.18, 0.37]	0.17	0.66	0.97
A4	su26	96	45	30	—	—	—	0.96

from $r = 0.28$ to 0.85, and their size depends on how much room the curated rate has to vary rather than on anything about the mutants. A kill rate pinned near its ceiling leaves little variation to correlate with. The highest correlation is on A2 ($r = 0.85$), the only assignment whose curated mean sits well below ceiling (0.81); on the other three, where students kill 0.95 to 0.97 of curated mutants on average, the correlations are both lower and inconsistent (0.28 to 0.62). Pearson r also exceeds Spearman ρ on every assignment, so what agreement they show comes from the weakest submissions rather than from consistent ordering. They do not establish mutant-to-mutant coupling, common fault behavior, or behavioral overlap between the two sets.

These observations do not show that operator mutants are unusable for grading. They show what an instructor must still decide, and our deployed curated set records those decisions. This study does not compare them with another instructor's curation of the operator mutants.

5.2 RQ2: What Suites Miss Among Our Curated Mutants

To describe what student suites miss, one author classified each curated mutant from its approved seed phrase into one of three buckets. A *happy-path* fault is one a typical test already hits. A *specific- or complex-input* fault appears only for an input a routine test omits. A *strong-assertion* fault is reached by ordinary tests but detected only by checking a property such as an exception type, a priority level, or preserved metadata. The classification is descriptive and used no second rater. The per-mutant labels are in the supplemental material. Table 3 reports how many curated mutants fall in each bucket, with the highest in-bucket survival rate.

Table 3. What student suites miss among our curated mutants on each assignment (Spring 2026): curated mutants per *why*-bucket, with the highest in-bucket survival rate. The hardest cell in each assignment is **bold**; per-mutant labels are in the supplement.

Why a test misses the mutant	A1	A2	A3	A4
Happy-path (a typical test hits it)	16 (8%)	4 (8%)	4 (15%)	1 (1%)
Non-obvious: specific/complex input	10 (3%)	**15 (48%)**	11 (8%)	21 (6%)
Non-obvious: strong assertion	**2 (15%)**	2 (28%)	**17 (23%)**	**14 (10%)**

The hardest bucket is the same on three of the four assignments: On A1, A3, and A4 it is strong assertion. Each of the three is covered by a routine test but goes undetected because the test does not check the contract property.

A1's strong-assertion bucket holds two mutants that encode the same fault in two ingredient classes. Both make `toString()` include the ingredient's free-text notes, which the displayed form must omit (15% Spring active survival each). Detecting either requires constructing an ingredient that actually carries notes and asserting on the whole rendered string. A test that leaves the notes field empty, or that checks only the amount and unit, calls `toString()` without ever exposing the leak.

A3's hardest mutant is `RC9_FindRecipeByIdWrong`, at 23% Spring active survival. It makes `findRecipeById` return the wrong recipe. Detecting it requires adding *multiple* recipes to a collection, calling `findRecipeById`, and asserting that the returned recipe is the right one. Students who missed it typically asserted only that the result was not `null`.

A4's hardest mutant drops an ingredient's preparation and notes during conversion, so a converted "sifted" flour comes back as plain flour (10% Spring active survival). A test detects it only by converting an ingredient that carries such metadata and asserting that those fields survive. A test that checks only the converted amount and unit exercises the conversion without observing the loss.

A2 is the exception. Its hardest mutant, `ST3_WrongRulePriority` (48% Spring active survival), registers a recipe's own conversion rules at `STANDARD` priority instead of `RECIPE`, so they no longer outrank the general rules they exist to override. Detecting it requires a registry holding two rules that disagree for the same conversion, one at each priority, and an assertion that the recipe-level rule wins. Given a single applicable rule the conversion returns the same answer either way. What the suite lacks is a second, conflicting rule in its input, not a stronger check on the output. Figure 6 traces this mutant from its spec rule to the hint a student received.

A2 also has the lowest curated kill rate of the four, 0.81 in Spring and 0.83 in Summer against 0.95–0.97 elsewhere

Table 4. Summer 2026 A2 hint deployment: adoption and cost. Every active student whose submission left a mutant undetected saw a seed phrase inline; the figures below are for the on-demand LLM expansion. Token counts and list-price cost are detailed in §6.

Active A2 students	113
Requested ≥ 1 expansion	52 (46%)
Expansions served	261 (2.31 per active student)
Input / output tokens	2.37M / 46K
List-price cost	$7.81 (≈ $0.03 per expansion)

(Table 2), and it is the first assignment whose tests must check substantive behavior rather than domain model construction. Its 21 mutants cover five graded units, and the same set ran in both cohorts.

Survival rates span almost the full range, and the ranking is stable across cohorts. Survival runs from 0.3% to 48% in Spring and from 0% to 39% in Summer, with the same two mutants easiest in both, `SC4_LosesRangeStructure` and `SC3_InvertedScaleFactor`, each killed by 94–100% of active submissions. The same three were hardest in both: `ST3_WrongRulePriority`, `CI6_WrongPriorityWith Ingredient`, and `CI4_WrongExceptionType`. A mutant is hard when no happy-path test forces the contract it violates. Fault depth does not predict difficulty.

5.3 RQ3: Deploying Mutant-Grounded Hints

We deployed the LLM expansion in Summer 2026 on A2. Every student saw the instructor-approved seed phrase inline at each failing mutant unit in the grading interface. The expansion elaborates that phrase against the specification on request. Students chose when to request an expansion, so every association reported below is observational and none of it identifies an effect caused by the expansion.

Of the 113 active A2 students, 52 (46%) requested at least one expansion, for 261 requests in total (2.31 per active student; Table 4). The expansions cost about three cents each ($7.81 for all 261, §6), bounded by the specification length rather than submission size.

Our unit of analysis is a *submission–graded-unit pair*: one student's A2 submission and one of the five mutant-graded units that the submission had not yet fully passed. Seed phrases are authored per mutant, but a hint is requested and delivered per graded unit (e.g., "write tests for ingredient conversion"). The action inlines the surviving mutants' phrases into that unit's failing-test output, and the expansion is generated from that output as a whole, so one request can address more than one mutant. Points are awarded per unit as well. The graded unit is therefore both the level at which a student acts and the level at which improvement is observable. A pair improved when the points earned on that unit increased on the student's next A2 submission. We classified each pair at the first submission as *hinted* when the student requested an expansion for that unit, *spillover* when the student requested an expansion for another unit on the same submission, and *no request* when the student requested no expansion on that submission.

Of the 261 expansions served across A2, 200 requests concerned these five mutant-graded units (the other 61 requests were for bugs in implementations, not tests). The outcome analysis retained the 195 requested-unit pairs that had a next submission on which improvement could be observed. Hinted pairs improved at 59.0% (115/195), compared with 36.3% (934/2,571) for no-request pairs. The risk ratio was 1.62; because a student contributes multiple pairs, the modified-Poisson model clustered standard errors by student (95% confidence interval [1.34, 1.97]).

Starting credit is the percentage of the graded unit's available points earned on the first submission in the pair. It was higher for hinted than no-request pairs (median 67% vs. 0%). Adjusting for this measured difference gave RR 1.65 (95% CI [1.37, 2.00]). The descriptive within-stratum RRs were 1.58 at 0%, 3.30 at 1–49%, and 1.34 at 50–99%.

Spillover pairs bear on the most obvious alternative explanation, that the students who requested an expansion (*requesters*) were simply the more diligent students. These are units the same students left unfixed on the very submissions for which they requested an expansion elsewhere. They improved at 26.3% (88/334), below the hinted pairs drawn from those same submissions and below the no-request pairs from students who asked for nothing. What distinguishes hinted pairs is therefore not which students sought help, though it may still be which units they chose. Hinted pairs also started with more credit than spillover pairs (median 67% vs. 25%), so students may have asked about the units they were closest to fixing. Adjusting for starting credit does not remove unmeasured differences in attention, motivation, difficulty, or intent to revise.

The spillover check works within a student. The between-student version asks whether the requesters were simply the stronger ones. Averaged across all of a student's pairs on these same five units, including the ones they did *not* ask about, improvement was similar for the 51 requesters and the 55 non-requesters (42.8% vs. 41.1%, Wilcoxon $p = 0.74$). Both models run over the same 3,100 pairs. That is not in tension with the pair-level association of 1.62, which concerns only the particular *units* a student selected for a request. Those units are a small share of what a student-level average covers: Requesters asked about roughly a tenth of their eligible pairs (200 of 1,957), so a per-unit association of this size barely moves the student average. Neither comparison establishes that the two groups had equal ability or engagement.

The Spring cohort had a different resource for the same failures, the instructor-pinned threads described in §5. Their uptake is worth recording. Among failing pairs where a

thread was served, pairs whose thread was opened improved *less* often than pairs whose thread went unread, 49.9% (202/405) against 58.1% (86/148) on A2 (RR 0.86, 95% CI [0.73, 1.02]) and 31.0% against 45.0% on A3 (RR 0.69, 95% CI [0.63, 0.75]).

That direction is the reverse of the hint result, and the likely reason is that the two channels select differently. A standing thread is opened when a student is already stuck, whereas an expansion is requested while revising the unit it concerns. The channels also differ in content, coverage, and cohort, so we report these figures as prior course context and draw no comparison with the expansion results.

Students used the expansions, and the units they asked about improved more often than the units they did not. What the deployment does not settle is why they chose those units. The spillover pairs rule out the explanation that requesters were simply the stronger students, but not the explanation that they asked about the units they were already closest to fixing. §6 takes up what that leaves open.

6 Discussion and Limitations

6.1 Prompt and Model Limits

The formative corpus was scored by one reviewer using a rubric developed during scoring, without a second coder. It did not include the deployed model, and we did not rubric-score the deployed hints (§4). The prompt guardrails therefore document engineering responses to observed failures, not a quality guarantee. Claude Sonnet 4.6 is hosted and may change or be retired; we record the model string and version the prompt, but cannot make the hint layer as reproducible as the PIT analysis. We did not compare the LLM expansions with instructor-written expanded hints.

6.2 Elicitation Scope and Instructor Effort

The LLM proposes misreadings before the instructor reviews them. Its proposals can anchor what the instructor considers, and review does not remove that bias. Moreover, asking for *specification misreadings* favors explicit contract errors over faults caused by poor modeling, tangled control flow, mis-wiring, or other implementation processes. The omissions may affect students differently, including those working across language or accessibility barriers. The kill-gate establishes that an approved mutant is observable to the instructor suite, not that the curated set represents all student faults. Thus, RQ2 describes difficulty *among our curated mutants*; it partly reflects what the curator chose to include. A future process could have instructors sketch faults before seeing LLM proposals and audit the result against learning objectives and prior student errors.

In retrospect, the participating instructor estimates about one hour per assignment for this reviewed loop. The same instructor recalls about one hour per assignment screening operator mutants: judging which seemed easy or hard, likely equivalent or coupled, and assigning points. The instructor had discontinued fully hand-authored semantic faults as too time-consuming. We have no time logs, matched mutant sets, or contemporaneous manual baseline. These recollections provide practitioner context, not evidence that the LLM workflow is more efficient or produces equivalent-quality mutants. We did not compare its mutants with a manually authored set in the same setting. No prior work we found reports a per-assignment or per-mutant authoring time for curated mutants, so these recollections cannot be checked against a published baseline. The nearest published anchors measure different quantities: one month of work from two experts for the misconception analysis behind a conceptual-mutant set [21], and a study confined to four assignments because authoring adversarial implementations was "potentially time-consuming" [30]. Outside mutation testing, per-item authoring times have been logged for other instructional artifacts, including problems with scaffolded hints in a template-driven tutoring-system builder [23, 28] and cluster-level feedback messages produced by program synthesis [8]. The artifacts and the tooling differ enough from curated semantic mutants that we do not read those figures as a baseline.

6.3 Live Generation Versus Reviewed Static Hints

Because each call receives only the specification and an approved seed, the hint expansions could have been statically generated and reused for each request. The Spring channel contained hand-written expansions of this kind for a handful of units (§5.3), so the static alternative is demonstrably feasible. Static or cached hints would reduce nondeterminism, provider dependence, and the risk of an unreviewed output. We chose an architecture that supports live generation because the same service also expands hints for failing *implementation* units, where the input is that submission's exception message and stack trace. Such input cannot be written in advance or reused across students, so a static design would not serve it. Those implementation hints are outside the scope of this paper, which concerns only the mutant-grounded expansions.

The deployed 261 expansions consumed 2.37M input and 46K output tokens, about 9,100 input and 180 output tokens per hint. At the model's then-current list price of $3 and $15 per million input and output tokens, respectively, the total was about $7.81, or three cents per expansion; a research credit made them free to us. Input is bounded by the specification rather than submission size. Caching by specification version and seed would further reduce calls.

6.4 Self-Selection and Outcome Measurement

Requesting a hint for a given graded unit signals attention to it and an intent to revise it. Adjusting for starting credit and using student-clustered standard errors address measured

starting credit and repeated pairs, not unmeasured motivation or task difficulty. "Improvement on the next submission" is also a coarse outcome. Students resubmit for many reasons and may use external LLMs that we cannot observe. The intensive six-week Summer cohort may differ in availability and motivation from other students. RQ3 therefore reports adoption and an observational pair-level association, not a causal hint effect.

6.5 Retrospective Analysis Design

The operator–curated analysis was implemented retrospectively on retained Spring submissions in three successive ways; these are analysis designs, not classroom conditions. First, default PIT mutated each student's production source, which conflated implementation differences with test-suite quality. Second, it mutated one fixed retained solution snapshot for all submissions, aligning the source families but ignoring canonical revisions during the term. Third, for A2–A4, it mutated the exact grader-time canonical commit recorded for each submission. A1 lacks grader-commit metadata and uses the fixed-snapshot fallback. About 5% of submissions were excluded because they did not compile against the canonical implementation, generally because their tests used student-defined APIs; the reported means may therefore run high. Aggregate kill-rate correlations cannot establish fault-level coupling or behavioral overlap.

6.6 Adoption and Portability

One author both teaches CS 3100 and co-develops Pawtograder, and the workflow has not been exercised by an instructor who did not build it. The evidence comes from one institution, one course, and the Java/JVM platform. An adopter needs the PIT fork[1], a versioned solution repository containing the specification, reference, mutants, and approved seeds, and a credentialed hint server. PreBakedMutator's constant-pool rewrite is JVM-specific and would need a new implementation for another runtime.

7 Conclusion

We presented and deployed a human–LLM workflow for creating two reviewed representations of a semantic fault: a whole-class mutant for grading a student test suite and a seed phrase for generating an on-demand hint. PreBakedMutator runs the curated mutants through PIT, while the hint service expands the paired phrase against the assignment specification without student code in context. A same-code comparison characterized how the operator and curated mutants differed in number, reachability, and their per-submission kill-rate associations.

In the deployment cohort, 46% of students requested at least one expansion. Failing submission–unit pairs with a self-selected request improved on the next submission at 1.62 times the rate of no-request pairs, at roughly three cents per expansion. This association does not establish that the hints caused the improvement. We also did not compare LLM-assisted curation with manual mutant authoring, or LLM-expanded hints with instructor-written expanded hints.

It therefore remains open whether the workflow reduces instructor effort, preserves the quality of fully manual artifacts, or transfers to an instructor who did not build the pipeline. Other open questions include portability beyond the JVM and courses whose specifications leave less room for the misreadings these mutants encode. We release the curated mutants, prompts, tools, and analysis so others can investigate those questions.

Data availability. The supplementary material is archived at https://doi.org/10.5281/zenodo.22085289. It includes the four CookYourBooks solution repositories (specification, reference implementation, instructor tests, whole-class mutant source variants, and each `autograder.yml` seed-hint and grading configuration), a source snapshot of the PreBakedMutator PIT extension, the production hint prompt (`promptData.ts`), the formative rubric and corpus, the operator/curated/counterexample PIT harnesses, the error-pin-uptake bundle, the aggregate CSVs behind the reported quantitative results, and the pipeline that derives them. The raw Pawtograder exports are per-student data and are withheld, so the deposit supports checking the reported numbers against the aggregates rather than rerunning the analysis end to end. The CS 3100 course materials are published at https://neu-pdi.github.io/CS3100-Spring-2026/ (source: https://github.com/neu-pdi/CS3100-Spring-2026). Pawtograder is open source at https://github.com/pawtograder/platform, documented at https://docs.pawtograder.com.

Acknowledgments

We thank Daniel Patterson, whose prior FeedBot work provided the architectural starting point for the system described here. We also thank all of the students, TAs and co-instructors who used the system and provided feedback on these assignments. This research was reviewed and approved as IRB #26-05-07 by the Northeastern University Division of Human Subject Research. This work was supported in part by a Khoury College Research Apprenticeship. This paper, its analysis code, and its tool implementation were prepared with assistance from Anthropic's Claude, used via Claude Code. The authors directed that work, verified every reported number against the underlying data, and are responsible for the content.

References

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd ed.). Cambridge University Press, Cambridge, UK.

[1] https://github.com/pawtograder/pitest; a source snapshot is also included in the supplementary material.

ISBN 9781107172012.

[2] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[3] Renzo Degiovanni and Mike Papadakis. 2022. μBERT: Mutation Testing Using Pre-Trained Language Models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, Piscataway, NJ, USA, 160–169. https://doi.org/10.1109/ICSTW55395.2022.00039

[4] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[5] Molly Q. Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards Answering "Am I on the Right Track?" Automatically Using Program Synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3358711.3361626

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing* (2nd ed.). MIT Press, Cambridge, MA, USA. ISBN 978-0-262-53480-2.

[7] Braxton Hall and Elisa Baniassad. 2022. Evaluating the Quality of Student-Written Software Tests with Curated Mutation Analysis. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2022)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3563767.3568132

[8] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S 2017)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[9] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Ángel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITiCSE-WGR '15)*. Association for Computing Machinery, New York, NY, USA, 41–63. https://doi.org/10.1145/2858796.2858798

[10] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[11] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 433–436. https://doi.org/10.1145/2610384.2628053

[12] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[13] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. CodeAid: Evaluating a Classroom Deployment of an LLM-based Programming Assistant That Balances Student and Educator Needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, Article 650, 20 pages. https://doi.org/10.1145/3613904.3642773

[14] Shriram Krishnamurthi, Thore Thießen, and Jan Vahrenhold. 2025. Porpoise: An LLM-Based Sandbox for Novices to Practice Writing Purpose Statements. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2025)*. Association for Computing Machinery, New York, NY, USA, 75–89. https://doi.org/10.1145/3758317.3759683

[15] Mark Liffiton, Brad E. Sheese, Jaromir Savelka, and Paul Denny. 2023. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research (Koli Calling '23)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3631802.3631830

[16] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas W. Price. 2019. The Impact of Adding Textual Explanations to Next-step Hints in a Novice Programming Environment. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 520–526. https://doi.org/10.1145/3304221.3319759

[17] Megan Maton, Gregory M. Kapfhammer, and Phil McMinn. 2026. Empirically Comparing Hazard-Guided LLM Mutation Techniques with Existing LLM- and Rule-Based Approaches. In *Proceedings of the 30th International Conference on Evaluation and Assessment in Software Engineering* (Glasgow, Scotland, United Kingdom) *(EASE 2026)*. Association for Computing Machinery, New York, NY, USA.

[18] James Perretta, Andrew DeOrio, Arjun Guha, and Jonathan Bell. 2022. On the Use of Mutation Analysis for Evaluating Student Test Suite Quality. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 263–275. https://doi.org/10.1145/3533767.3534217

[19] James Perretta, Andrew DeOrio, Arjun Guha, and Jonathan Bell. 2025. Instructor-Written Hints as Automated Test Suite Quality Feedback. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE TS 2025)*. Association for Computing Machinery, New York, NY, USA, 910–916. https://doi.org/10.1145/3641554.3701866

[20] pitest.org. 2026. Mutation Operators. https://pitest.org/quickstart/mutators/ Accessed 2026-06-11.

[21] Siddhartha Prasad, Ben Greenman, Tim Nelson, and Shriram Krishnamurthi. 2024. Conceptual Mutation Testing for Student Programming Misconceptions. *The Art, Science, and Engineering of Programming* 8, 2, Article 7 (2024). https://doi.org/10.22152/programming-journal.org/2024/8/7

[22] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Petersen, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots Are Here: Navigating the Generative AI Revolution in Computing Education. In *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR 2023)*. Association for Computing Machinery, New York, NY, USA, 108–159. https://doi.org/10.1145/3623762.3633499

[23] Leena Razzaq, Jozsef Patvarczki, Shane F. Almeida, Manasi Vartak, Mingyu Feng, Neil T. Heffernan, and Kenneth R. Koedinger. 2009. The ASSISTment Builder: Supporting the Life Cycle of Tutoring System Content Creation. *IEEE Transactions on Learning Technologies* 2, 2 (2009), 157–166. https://doi.org/10.1109/TLT.2009.23

[24] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[25] Marie Salomon, Kyle D. Chin, Reid Holmes, Thomas Fritz, and Gail C. Murphy. 2025. An Exploration of How Generative AI Affects Workflow and Collaboration in a Software Engineering Course. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2025)*. Association for Computing Machinery, New York, NY, USA, 42–54. https://doi.org/10.1145/3758317.3759680

[26] Zalia Shams and Stephen H. Edwards. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/2493394.2493402

[27] Frank Tip, Jonathan Bell, and Max Schäfer. 2025. LLMorpheus: Mutation Testing Using Large Language Models. *IEEE Transactions on Software Engineering* 51, 6 (2025), 1645–1665. https://doi.org/10.1109/TSE.2025.3562025

[28] T. E. Turner, M. A. Macasek, G. Nuzzo-Jones, N. T. Heffernan, and K. Koedinger. 2005. The Assistment Builder: A Rapid Development Tool for ITS. In *Proceedings of the 12th International Conference on Artificial Intelligence in Education (AIED 2005)*. IOS Press, Amsterdam, The Netherlands. http://web.cs.wpi.edu/Research/trg/public/project/papers/AIED2005/builderAIED.doc

[29] John Wrenn and Shriram Krishnamurthi. 2019. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. Association for Computing Machinery, New York, NY, USA, 131–139. https://doi.org/10.1145/3291279.3339416

[30] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18)*. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999

[31] Lucas Zamprogno, Reid Holmes, and Elisa Baniassad. 2020. Nudging Student Learning Strategies Using Formative Feedback in Automatically Graded Assessments. In *Proceedings of the 2020 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2020)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3426431.3428654

[32] Elaine Zhu, Smaran Teja, Chris Coombes, and Daniel Patterson. 2025. FEEDBOT: Formative Design Feedback on Programming Assignments. In *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Nijmegen, Netherlands) *(ITiCSE 2025)*. Association for Computing Machinery, New York, NY, USA, 576–582. ISBN 9798400715679. https://doi.org/10.1145/3724363.3729063