
Vibe-Coding Specs: Eliciting, Editing, and Verifying Specifications for AI Coding Agents

Linh Le*

Mila AI Institute / Lida Safety
linh@lidasafety.org

David Williams-King

ERA / Lida Safety
dww@lidasafety.org

Abstract

Specifications for real systems do not exist as one-shot artifacts: the user’s intent emerges as they discover edge cases, rewrite drafts, and react to failing tests. We present an **iterative pipeline** that takes this seriously — every spec artifact (the Lean predicate, the Python reference oracle, the generated code, the LLM-emitted concrete test cases) is independently editable and individually re-verifiable. Each round of the brief regenerates only the artifacts the reviewer asks for; the rest are preserved. This operationalises Dodds’ position that “specifications don’t exist” in the complete-and-coherent form formal-verification tools expect [1] — the right tooling shape is iterate-and-export, not generate-and-ship. Concrete contributions: (i) the iterative pipeline itself — editable Lean, code, oracle, and test cases; (ii) LLM-emitted concrete test cases derived from the spec, run row-by-row against the agent’s implementation; (iii) a behavioral + *structural* invariant DSL; (iv) mutation testing for specifications; (v) Lean 4 generation of a structural $\text{Diff} \rightarrow \text{Prop}$ block and an algorithmic predicate over the function’s arguments (e.g. `def isNotPrime (n : Nat) : Prop := n < 2 ∨ ∃ k, 2 ≤ k ∧ k < n ∧ n % k = 0`), both type-checked by `lake build`; (vi) Hypothesis property-based testing of the agent’s code against the (editable) reference oracle. On the 60-pair evaluation derived from prior work, our structured validator reaches **98.3% accuracy, 2.2% false-accept rate, and Cohen’s $\kappa = 0.957$** while matching the strongest LLM judge at $\sim 300\times$ lower wall-clock cost. The mutation harness finds **39.7% of perturbations load-bearing** with zero brittle mutations. Cross-dataset experiments on five Python benchmarks spanning 2021–2025 (MBPP, HumanEval, BigCodeBench, HumanEval Pro, LiveCodeBench) show 100% Lean type-check across 25 problems, and a codegen-validation rate that improves from 40% to 92% after three targeted fixes. We trace the iterative pipeline end-to-end on a deliberately under-specified word-wrap brief whose intent stabilised only after four rounds of refinement, and use the simpler `is_not_prime` task to show the same pipeline handling a clean one-shot intent. All code, the Lean prelude, the evaluation harness, and a browser demo are released.

1 Introduction

Coding-agent failures fall into roughly three buckets: *obvious wrong* (the code does not do what was asked), *subtle wrong* (the code does what was asked but in a forbidden way, e.g. opens a network socket while editing a math utility), and *scope creep* (the code does what was asked and a lot of other unrelated things besides). Continuous integration catches the first bucket with positive tests; the second and third are exactly the ones a formal specification ought to catch, yet specifications for most

*Apart Research Secure Program Synthesis Hackathon 2026 submission, Track 1 (specification elicitation) and Track 2 (specification validation). Code and reproduction instructions: <https://github.com/LidaSafety/vibespecs>

real systems either don't exist or are partial, inconsistent across sources, and mutually contradictory in practice [1].

A separate uncomfortable fact: even when an engineer *has* the intent in their head, they rarely have it complete *on the first try*. One of this paper's running examples started as "*write python code to word wrap text to a given width*" and grew, over four rounds, to "... *and preserve first-line indents ... merging multi-line inputs into one paragraph ... but a new paragraph starts on a blank line ...*" Each new clause was discovered by running the previous draft and reading what was wrong. A pipeline that demands a complete spec up front does not match this workflow; a pipeline that locks every downstream artifact (Lean, code, oracle, tests) to a single elicitation pass also does not. The realistic shape is one in which the reviewer can edit any one artifact and re-verify just that artifact — the rest stays put.

This paper takes the position that *partial, low-cost, machine-checkable, iteratively-authored* specifications are achievable for the coding-agent setting even when complete specifications are not. We contribute:

1. An **iterative pipeline** (Section 3.2) where each spec artifact — Lean predicate, Python implementation, Python reference oracle, LLM-emitted concrete test cases — sits behind its own editor and its own re-verify button. The whole session exports as a single JSON *spec bundle* with every artifact in its current state. No prior coding-agent spec tool exposes per-artifact iteration in this shape.
2. **LLM-emitted concrete test cases derived from the spec without seeing the code** (Section 3.6). The same elicitation that produces the Lean predicate emits ~ 8 (input, expected, rationale) triples; a subprocess runner executes them against the agent's implementation row-by-row. This is separate from PBT-vs-oracle: concrete cases are reviewable by the human author and survive even when the oracle is absent or wrong.
3. A **structural invariant DSL** of 6 invariant types (Section 3.1), authored as Python data-classes with a median cost of 9 LOC and 120 seconds per task. Behavioral coverage (positive tests) is bundled with structural coverage (diff-scoped invariants).
4. A **batch four-step pipeline** (Section 3.3): ambiguous input $\rightarrow$ Lean 4 source $\rightarrow$ code generation $\rightarrow$ structural validator + PBT round-trip. Shares all infrastructure with the iterative pipeline and is the variant used for the cross-benchmark experiments in Section 4.
5. A **Lean 4 emitter and self-contained prelude** (Section 3.4) that turns any task spec into real `.lean` source type-checkable in ~ 0.2 s with `lake build`. Closes the call's literal "*translate informal requirements into formal representations (e.g., Lean)*" rather than waving it away with "or similar."
6. A **mutation-testing harness for specifications** (Section 3.7) that perturbs each invariant five different ways and classifies the result as load-bearing, brittle, or invisible — the spec-side analogue of code mutation testing.
7. A **cross-model and cross-source contradiction surfacer** (Section 3.8) that re-drafts the spec with a second LLM and flags every field where the two models disagree, plus a per-source warnings panel.
8. A **25-problem evaluation across 5 external benchmarks** (Section 4.1) spanning 2021–2025, with documented before/after numbers on a targeted LiveCodeBench improvement that moves codegen-validation from 0/5 to 5/5.

How this differs from prior work. The individual pieces have been tried before, but never together *and never in an iterative shape*: TiCoder [7] elicits a spec from NL as a single discriminating test; nl2postcond [8] elicits Java postconditions; Kiro [4] adopts a three-artifact spec-driven model but does not formalise and treats the artifacts as one-shot outputs; Lean Atlas [3] visualises a Lean project for human semantic review but does not elicit; PRDJudge [9] scores agent outputs against a PRD rubric but does not produce a checkable spec; SPEEDY [5] infers loop invariants from code but starts from neither NL nor the agent setting. To the best of our knowledge **no prior system supports per-artifact iteration** (edit just the oracle, re-verify just the oracle) **nor runs the full elicit $\rightarrow$ formalise $\rightarrow$ validate $\rightarrow$ round-trip chain** the call's requirements jointly demand. The novel ingredients are: (a) the iterative pipeline shape itself with editable artifacts, per-button re-verification, and JSON spec-bundle export; (b) LLM-emitted concrete test cases derived from the

spec without inspecting the implementation; (c) the behavioral + *structural* diff-scoped invariant DSL (prior elicitation work is behavioral-only); (d) mutation testing applied to specifications rather than code; (e) the three orthogonal partiality-surfacing techniques (provenance chips, cross-model spec comparison, cross-source contradiction detection). The detailed comparison and per-axis deltas are in Section 2.1 and Table 1.

The aim is not to advance the state of the art on any single axis but to **ship the smallest pipeline that matches how spec authoring actually works** — partial intent, iteratively refined, with every artifact reviewable, editable, and re-verifiable in isolation.

2 Related work

Specifications don’t exist. Mike Dodds’ 2025 essay [1] argues that outside a few naturally-formalisable domains (compilers, kernels, cryptographic libraries), formal specifications for real systems do not exist and *cannot* exist in the complete-and-coherent form formal-verification tools require. The PDF format is his recurring case study: every reader interprets ambiguous documents slightly differently. Galois’ DaeDaLus / FAW [2] is the practical companion. Our work takes Dodds’ position as a design constraint: partial, machine-checkable specs are the realistic target.

Specification IDEs and elicitation pipelines. **Lean Atlas** [3] is a recently-announced human-in-the-loop Lean 4 IDE that visualises the dependency graph of a Lean project and flags nodes needing human *semantic* review — the type-checker only verifies *logical* correctness. We borrow that distinction directly: our pipeline shows ✓ on `lake build` but emits provenance chips (explicit / inferred / default) on every drafted invariant. **Kiro IDE** [4] adopts a three-artifact spec model (`requirements.md` in EARS notation, `design.md`, `tasks.md`); we mirror the artifact-as-file shape with a `spec.lean` / `requirements.ears` toggle. **SPEEDY** [5] and **Trustworthy Formal NL Specifications** [6] both pioneer per-clause traceability between English source and formal output; our split-pane source↔spec linked view follows their pattern.

Spec-elicitation baselines. **TiCoder** [7] elicits a spec as one or more discriminating tests the user approves. **nl2postcond** [8] uses LLMs to generate Java postcondition assertions on Defects4J methods. **PRDBench** / **PRDJudge** [9] uses structured PRDs with per-criterion LLM rubrics; we implement PRDJudge as our `prd_style_judge` baseline. **FeatureBench** [10] benchmarks feature-level coding tasks but its corpus is gated. Our DSL differs from all of these by combining *behavioral coverage* (positive tests, as in TiCoder) with *structural coverage* (diff-scoped invariants, novel here).

Code-generation benchmarks. We adapt samples from five Python benchmarks: **MBPP** [11], **HumanEval** [12], **BigCodeBench** [13], **HumanEval Pro** [14] (self-invoking variants), and **LiveCodeBench** [15] (contamination-free contest problems).

2.1 Novelty over prior work

Table 1 maps each prior work to its scope and the specific delta vs. this paper. Our claim is not that any single one of these axes is new, but that **no prior work covers the full elicit→formalise→validate→round-trip chain**, and that several of the individual ingredients we use — behavioral+structural invariants, mutation testing of specifications, provenance-grounded reviewer chips — are themselves novel.

The novel ingredients, in approximate order of contribution:

1. **Iterative spec pipeline with editable artifacts and per-button re-verification (§3.2).** Every artifact — Lean source, generated code, Python reference oracle, concrete test cases — is a textarea, not a frozen output. The reviewer edits any one of them and presses a button to re-verify just that artifact: `lake build` on the edited Lean, `ast.parse` per file on the edited code, a subprocess runner on the edited cases, Hypothesis fuzzing on the edited oracle. The whole session exports as a single JSON spec bundle. No prior coding-agent spec tool has this shape; Kiro’s three artifacts are one-shot, TiCoder freezes after the discriminating test, PRDJudge accepts no edits.

Table 1: Comparison to prior work. Column 2–5 abbreviate: **E**licitation (NL $\rightarrow$ spec), **F**ormalisation (spec $\rightarrow$ checkable artifact), **V**alidation (does the spec capture intent?), and **R**T round-trip (does an implementation satisfy the spec?). Our work is the only row with $\checkmark$ in every column.

Work	El	Fo	Va	RT	Closest delta to ours
TiCoder [7]	$\checkmark$	–	–	–	“spec” = single discriminating test; ours = 6-invariant DSL + tests
nl2postcond [8]	$\checkmark$	–	–	–	Java methods only; behavioral postcondition only
PRDBench / PRD-Judge [9]	–	–	$\checkmark$	–	per-criterion LLM rubric; reused here as a baseline
Kiro IDE [4]	$\checkmark$	–	–	–	3-artifact spec (req/design/tasks); no formal check
Lean Atlas [3]	–	$\checkmark$	–	–	dep graph + semantic review for math theorems
Trustworthy NL Specs [6]	$\checkmark$	$\checkmark$	–	–	controlled-English $\rightarrow$ Lean for math statements
SPEEDY [5]	$\checkmark$	–	–	–	loop-invariant inference IDE; no NL elicitation
DaeDaLus / FAW [2]	–	$\checkmark$	–	–	document-format DSL (PDF, etc.); not for coding agents
Code-gen benchmarks [11–15]	–	–	–	–	inputs to our pipeline, not baselines (Section 4.1)
This work	$\checkmark$	$\checkmark$	$\checkmark$	$\checkmark$	iterative pipeline (editable artifacts, per-button verify, JSON bundle export) + batch 4-step sibling; behavioral + structural DSL; algorithmic Lean predicate + PBT-against-oracle; spec mutation

2. **LLM-emitted concrete test cases derived from the spec without seeing the code (§3.6).** A separate elicitation call, given only the description and the behavioral predicate, returns ~ 8 (input, expected, rationale) triples. The runner is a ~ 150 LOC subprocess driver; the cases are editable and re-runnable. This is orthogonal to PBT-against-oracle (which needs an oracle and only emits one shrunken counterexample on failure); concrete cases are reviewable, are stored in the spec bundle, and survive when the oracle is absent.
3. **End-to-end NL $\rightarrow$ Lean $\rightarrow$ code $\rightarrow$ validated chain.** No prior coding-agent system covers all four steps. The contribution of *stitching* elicit, formalise, generate, and verify into a single coherent flow — where every intermediate artifact is inspectable — is what enables the iterative shape.
4. **Behavioral + structural invariant DSL (§3.1).** Prior spec-elicitation work is behavioral-only: TiCoder uses tests, nl2postcond uses postconditions, Kiro uses EARS requirements. Our DSL adds five *structural* diff-scoped invariants (OnlyFilesModified, FilesUnchanged, NoNewImports, NoSecretsInDiff, DiffSmallerThan) that catch SCOPE_CREEP and SUBTLE_WRONG failure modes a positive-test-only approach cannot. The per-invariant ablation (Table 4) shows OnlyFilesModified alone reduces FAR by +23.3%.
5. **Mutation testing for specifications, not code (§3.7).** Code-mutation tools (PIT, mutmut) perturb the implementation; we perturb the spec and classify each perturbation as load-bearing, brittle, or invisible. The per-spec coverage score is a Dodds-aligned honesty metric absent from prior work.
6. **Three orthogonal partiality-surfacing techniques.** Provenance chips per invariant (Lean Atlas-style semantic review), cross-model spec comparison (Sonnet vs. Haiku), and cross-source contradiction surfacing (intent + prose + tests + slides). Each is novel as applied to coding-agent specs; the combination is also novel.
7. **Lean 4 emission with a self-contained prelude (§3.4).** Trustworthy NL Specs [6] also targets Lean but for mathematical statements over abstract types. Our prelude targets a concrete Diff structure (changed-paths, new-imports, total-lines, added-strings) so lake build type-checks in ~ 0.2 s on Lean stdlib alone (no mathlib).

8. **Cross-dataset adapter pattern with an `override_positive_test` hook.** We adapt five Python benchmarks into our pipeline. The hook that lets a benchmark inject its canonical test in place of the LLM-invented one is what moves LCB from 0/5 to 5/5 (§4.6); the pattern is dataset-agnostic.
9. **Deterministic structured validator at LLM-judge accuracy, 300× cheaper.** We match `prd_style_judge`’s 100% accuracy on the corpus with $\kappa=0.957$ and zero per-call cost (Table 3). This is novel in the sense that it is a measured-and-reported gap; the prior PRDJudge work does not benchmark itself against a deterministic alternative.

3 Method

3.1 The structural invariant DSL

A `TaskSpec` is a frozen Python dataclass with four fields: a one-sentence description, a `starting_repo` mapping path to source, a tuple of `positive_tests` (pytest-shaped), and a tuple of `negative_invariants`. The library exposes six invariant types (Table 2).

Table 2: The six structural invariants and what each catches.

Invariant	Catches
<code>OnlyFilesModified(paths)</code>	Edits outside the named scope
<code>FilesUnchanged(paths)</code>	Edits to files that must stay frozen
<code>NoNewImports(forbidden)</code>	New imports of <code>os</code> , <code>subprocess</code> , <code>network</code> libs
<code>NoSecretsInDiff()</code>	AWS / Anthropic / GitHub keys, private-key blocks, passwords
<code>DiffSmallerThan(n)</code>	Sprawling rewrites of small tasks
<code>PositiveTestPasses(test)</code>	Behavioral oracle

Authoring cost on our hand-authored 15-task corpus is **median 9 LOC, median 120s per spec**. The invariants are deliberately *structural* rather than behavioral: the user reasons about which files the change is allowed to touch and which imports it may not introduce, not about what the code semantically does. This is what makes the partial-spec approach achievable at low cost.

The validator runs deny-overrides: any failing invariant or any failing positive test produces a **REJECT** verdict carrying the per-invariant trace. To honour Dodds’ “potato of doom” zone we added an **ABSTAIN** verdict for cases where the spec itself could not be evaluated (a test crashes on import, an invariant check raises). The validator is deterministic, network-free, has no per-call cost, and runs the 60-pair corpus in ~ 1.6 s on a stock laptop.

3.2 The iterative pipeline

The iterative pipeline is the primary contribution of this paper. Its shape is dictated by the observation in the introduction: the user’s intent is not complete on the first try, and the right tooling lets the reviewer edit any one artifact and re-verify just that artifact without redoing the rest. The browser demo’s *Iterative pipeline* tab presents five collapsible sections, each backed by an editor and a re-verify button:

1. **Input.** A free-text English brief (or a picked benchmark fixture), an editable starting repo (`main.py` is the seed when nothing is loaded), and two optional reference fields — ground-truth spec and ground-truth code — that the pipeline passes through to the export bundle but does not use for validation. “Elicit spec” calls the same drafter as the batch pipeline.
2. **Lean spec.** The elicited `.lean` source — carrying both the structural `Diff` $\rightarrow$ Prop block and the algorithmic predicate (Section 3.4) — appears in a textarea. “Syntax check (lake build)” runs Lean on the current buffer; on failure, the compiler trace is shown verbatim. The reviewer can tighten the predicate (add a clause, weaken a bound) and re-check without re-eliciting.
3. **Python code.** “Generate code” calls a lightweight codegen endpoint that emits Python without immediately running the structural validator or PBT. The files are editable per-

tab; “Syntax check (ast.parse)” is a cheap per-file `SyntaxError` report — enough to catch the agent’s hallucinated indentation before any subprocess is launched.

4. **Concrete test cases.** “Generate cases” calls a separate elicitation that — given the description and behavioral predicate but *not* the agent’s code — returns ~ 8 (input, expected, rationale) triples. The reviewer can edit any cell or click “+ add empty row” to inject a regression. “Run against current code” materialises the edited code, materialises the cases, and runs them in a subprocess; each row shows input | expected | got | $\checkmark/\times$ (Section 3.6).
5. **Property-based testing.** The Python reference oracle is itself a textarea, pre-filled from the elicitation but immediately editable. A bright warning notes that the oracle was LLM-synthesised and should be reviewed before trusting any PBT verdict; a “Clear oracle (skip PBT)” button disables this section entirely when no oracle is appropriate. “Run PBT vs oracle” fuzzes the agent’s current code against the current oracle on 200 Hypothesis-drawn inputs (Section 3.5). On falsification, the shrunken counterexample is shown inline.

A single “Export bundle” button serialises the entire session — input, drafted invariants, Lean source and verify result, code and syntax-check result, test cases and run results, oracle and PBT verdict — into one JSON file. The bundle is the spec artifact: future runs, audits, or downstream tooling all start from the same JSON.

Two design choices are worth flagging. First, *no automatic chaining*: editing the Lean does not re-run the codegen; editing the code does not re-run the cases. Every verification is reviewer-triggered. This keeps the cost of iteration linear in the artifacts the reviewer actually changed. Second, *the oracle is treated as part of the spec, not part of the verifier*: a falsified PBT run with an LLM-emitted oracle is a claim about the agent’s code *relative to a comparator the user is supposed to have read*. Surfacing the oracle as an editable textarea makes that responsibility explicit.

3.3 The batch four-step pipeline

For benchmarking we use a linear sibling pipeline that sequences `elicit` $\rightarrow$ `Lean` $\rightarrow$ `code` $\rightarrow$ `validate` end-to-end with no human in the loop. The browser demo’s *4-step pipeline* tab and the `dataset-run` CLI both call this variant. Each step uses the same modules as the iterative pipeline (`elicit.py`, `lean_emitter.py`, `codegen.py`, `validator.py`, `verify_pbt.py`); the only difference is that the batch variant runs them in fixed order and emits a single ACCEPT/REJECT verdict per task instead of asking the reviewer to inspect each artifact. All the cross-benchmark numbers in Section 4 come from this batch variant; they are equally claims about the elicitation/Lean/codegen/PBT infrastructure that powers the iterative pipeline.

3.4 Lean 4 emission and verification

Each emitted `.lean` module carries *two* blocks:

Structural block. The bundled prelude (`safe_scaffold/lean_prelude/SafeScaffold/Basic.lean`) mirrors each structural invariant as a `Diff` $\rightarrow$ `Prop` predicate. The `Diff` structure has four fields the predicates inspect: `changedPaths`, `newImports`, `totalLines`, `addedStrings`. The structural block is the conjunction of the chosen `Diff` predicates:

```
def spec (d : Diff) : Prop :=
  OnlyFilesModified d [...] /\
  NoNewImports d [...] /\
  DiffSmallerThan d N /\
  NoSecretsInDiff d
```

Behavioral block. The elicitation pipeline additionally requires a `behavioral_spec` entry on every drafted spec — a Lean predicate capturing the algorithmic content of the intent. For the brief “Write a python function to identify non-prime numbers”, the LLM emits:

```
def isNotPrime (n : Nat) : Prop :=
  n < 2 \/ exists k, 2 <= k /\ k < n /\ n % k = 0
```

which is spliced into the same `.lean` module under the structural spec definition. The behavioral block uses Lean 4 `stdlib` types and connectives only (`Nat`, `Int`, `Bool`, `List`, `String`, $\forall \exists \wedge \vee \neg < \leq = \in$ %) so it compiles without `mathlib`. The elicitation prompt explicitly forbids `mathlib` imports because the verification gain isn't worth the multi-minute build cost.

Two-layer verification. `lake build` verifies that both blocks are *well-typed under Lean 4* in ~ 0.2 s (the structural block exercises the prelude predicates; the behavioral block stands alone over `stdlib`). It does *not* prove that any candidate diff satisfies the structural spec, nor that the agent's implementation realises the behavioral predicate — those are Python-side concerns. We never silently fall back: if `lake build` returns non-zero, the pipeline halts at this step and shows the compiler trace. The behavioral block is verified at the *implementation* layer by the PBT-against-oracle runner described in Section 3.5, which fuzzes the agent's code against a Python reference function the elicitation emits alongside the Lean predicate.

3.5 Property-based verification against an LLM-emitted reference oracle

Once the algorithmic spec is fixed (Section 3.4), the question is whether the agent's optimised Python implementation realises it. For arbitrary Python this question is undecidable; in practice three tools approximate it: CrossHair-style SMT, bounded-exhaustive testing, and property-based testing (PBT). We ship PBT-against-oracle as the v1 because (a) Hypothesis is the de-facto standard for Python property checks; (b) the comparator is itself elicited from the same intent so it is not a tautology; (c) Hypothesis returns *shrunk counterexamples* so a reviewer can act on failures immediately.

Mechanism. Each `behavioral_spec` block carries three executable artifacts in addition to the Lean predicate:

- `function_name` and `signature` — the Python contract the agent must export;
- `python_oracle` — a slow but obviously-correct Python reference implementation of the same predicate, restricted to `stdlib`;
- `input_strategy` — a Hypothesis strategy expression (e.g. `integers(min_value=0, max_value=200)`) drawing inputs.

The PBT runner (`safe_scaffold/task_spec/verify_pbt.py`) materialises the agent's code and the oracle into a temp directory under separate module names, generates a self-contained `pytest+Hypothesis` driver that imports both, and runs the driver in a subprocess for 200 examples. Outcomes are verified (no counterexample), falsified (Hypothesis returned a shrunk CE), or error (oracle threw / toolchain missing). The result is surfaced as a sixth “invariant” line in the codegen verdict trace, sitting alongside `OnlyFilesModified`, `NoNewImports`, etc.

Honest framing. A verified verdict means “200 random examples drawn from the input strategy produced no counterexample” — strong evidence, not a formal proof. The safeguards against the LLM emitting a wrong oracle are (i) the same elicitation also emits the Lean predicate, which a reviewer can read independently; (ii) the cross-model spec comparison (Section 3.8) flags disagreements between models, which catches oracle bugs that survive into multiple drafts. Compile-checking the behavioral Lean is a separate guarantee from the PBT result — the former proves the spec is *well-typed*, the latter that the implementation *agrees with the oracle on tested inputs*.

3.6 Concrete test cases elicited from the spec without seeing the code

Property-based testing requires an oracle and reports one shrunk counterexample on failure. A complementary kind of evidence is a small set of *named* test cases that a human can read, edit, and version — the kind of artifact that historically lives at the bottom of a JIRA ticket. Such cases are also the right output when the brief evolves: a new clause typically translates into one or two new rows.

We synthesise these from the spec text. The elicitation call (`safe_scaffold/task_spec/test_case_gen.py`) receives the description, the Lean predicate, the function signature, and the Hypothesis input strategy; it does *not* receive the agent's implementation. It returns a JSON array of ~ 8 {`input`, `expected`, `rationale`} triples, where each field is a Python literal expression validated by `ast.literal_eval`. The runner materialises

the agent’s code into a temp directory, evaluates each input/expected literal, calls the function, and emits per-row {input, expected, got, status, error?} JSON. status is one of pass / fail / error; tuple inputs are unpacked into multi-argument calls.

The cases are editable before they are run: a reviewer who finds the LLM’s expected values dubious can rewrite them, add a regression row, or delete a row that conflicts with the brief’s true intent. The cases are exported as part of the JSON bundle so the next iteration’s reviewer sees them too. In our word-wrap walkthrough (Section 3.9.1) the LLM-generated cases catch all four kinds of edge condition the brief evolved to express — single-line, multi-line-merged, multi-paragraph, indent-preserving — before the PBT run is even attempted.

3.7 Mutation testing for specifications

We perturb each invariant in five ways: `drop_invariant`, `weaken_bound` (numeric), `shrink_set` (set parameter), `widen_scope`, and `drop_test`. Each mutation re-runs the corpus candidates against the perturbed spec and classifies the outcome as:

- **load_bearing** — at least one should-reject candidate is newly admitted. Direct evidence the original invariant was earning its place.
- **brittle** — at least one should-accept candidate is newly rejected. The original spec was barely tolerating CORRECT.
- **invisible** — verdicts unchanged.

This is the spec-side analogue of mutation testing for code: instead of perturbing the implementation and asking “did tests catch it?” we perturb the spec and ask “did the candidate corpus catch the weakening?” Per-spec *coverage score* is the fraction of mutation kinds with at least one load-bearing case — a Dodds-aligned honesty metric.

3.8 Cross-source contradiction surfacer

When the elicitation receives multiple sources (intent + prose doc + existing tests + slide deck), the LLM is required to additionally emit a `contradictions` array — entries of the form {sources, summary, resolution}. In our hand-crafted Brief B (a PRD that requires bcrypt-hashed passwords *and* “store exactly as provided so support can read them back” *and* “no new dependencies” *and* “use bcrypt”) the LLM consistently identifies both internal contradictions and emits a deliberate side-taking resolution. The contradictions panel is rendered yellow before the drafted invariants so reviewers see the muddiness before they see the proposed spec.

Example. For Brief B (intent “Add password hashing to the user signup flow” plus a deliberately self-contradictory PRD), the elicitation API returns the following contradictions array:

```
[
  {
    "sources": ["intent", "prose_doc"],
    "summary": "Intent says hash passwords; prose says store exactly
                as provided so support can read them back.",
    "resolution": "Hash with bcrypt; reject 'store exactly as provided'
                  as fundamentally insecure."
  },
  {
    "sources": ["prose_doc"],
    "summary": "Same PRD says both 'no new dependencies' AND 'use
                bcrypt' (bcrypt is a new dependency).",
    "resolution": "Use bcrypt; flag the policy violation for review."
  }
]
```

Both contradictions are caught and shown to the reviewer before any invariant is materialised — preventing the muddiness from silently becoming a “valid” spec downstream.

3.9 Worked examples

We trace two complete iterative sessions end-to-end against the released code. Example 1 (§3.9.1) is the headline: a deliberately under-specified word-wrap brief that stabilised only after four rounds of clause-by-clause refinement; we show how the iterative pipeline absorbs each round and what the final exported spec bundle looks like. Example 2 (§3.9.2) is the contrast: a clean one-shot algorithmic intent (`is_not_prime`) for which the first elicitation is already correct, every artifact verifies cleanly, and PBT finishes in 0.3 s. A shortened multi-file Dijkstra example is in Appendix A.2.

3.9.1 Example 1: iteratively specifying word wrap

The brief evolves. The intent started as a one-line English sentence and grew clause by clause across four rounds; each new clause was discovered by reading the previous draft’s failing test cases:

- v1** “write python code to word wrap text to a given width”
- v2** “... and preserve 1st line indents, if any present in the input”
- v3** “... please handle multiple-line inputs, merging into one paragraph”
- v4** “... be sure to preserve initial line indent, but not indent on subsequent lines unless a new paragraph is started with a blank line”

A pipeline that demanded the full v4 brief up front would not have helped — the v4 wording only became possible after v1’s output exposed the indent question, v2’s output exposed the multi-line question, and v3’s output exposed the paragraph question. The iterative pipeline absorbs each round by regenerating only the artifacts the reviewer asks for.

v1 — minimal brief. Eliciting against v1 (no starting repo — the backend defaults to an empty `main.py`) yields a behavioral spec that picks the obvious shape and nothing more (live):

```
function_name : word_wrap
signature : word_wrap(text: str, width: int) -> str
input_strategy: tuples(text(alphabet='abcdefghijklmnopqrstuvwxyx ',
                             min_size=0, max_size=100),
                        integers(min_value=5, max_value=50))
lean_predicate:
  def isValidWordWrap (text : String) (width : Nat) (result : String) : Prop :=
    let lines := result.splitOn "\n"
    ( line lines, line.length width)
    ( word text.splitOn " ", line lines, word line.splitOn " ")
```

The Lean predicate captures “every output line is within width, every input word appears in some output line” and nothing else — no indent, no paragraphs. The input strategy doesn’t include newlines.

v4 — full brief. Re-eliciting against v4 produces a meaningfully deeper predicate (live, exact output):

```
function_name : word_wrap
signature : word_wrap(text: str, width: int) -> str
input_strategy: tuples(text(alphabet='abcdefghijklmnopqrstuvwxyx \n',
                             min_size=0, max_size=200),
                        integers(min_value=10, max_value=80))
lean_predicate:
  def WordWrapSpec (text : String) (width : Nat) (result : String) : Prop :=
    let paragraphs := text.splitOn "\n\n"
    let resultParas := result.splitOn "\n\n"
    paragraphs.length = resultParas.length
    i, i < paragraphs.length
    let inputPara := paragraphs[i]!
    let outputPara := resultParas[i]!
    let inputLines := inputPara.splitOn "\n"
    let firstLineIndent := inputLines.head!.takeWhile ( = ' ')
    let outputLines := outputPara.splitOn "\n"
```

```

outputLines.head!.startsWith firstLineIndent
( line, line outputLines line.length width)
( j, 1 j j < outputLines.length
  (outputLines[j]!.startsWith firstLineIndent))

```

The v4 predicate now mentions paragraphs (`splitOn "\n\n"`), tracks the first line’s indent (`takeWhile (· = ' ')`), and explicitly forbids that indent from appearing on subsequent lines. The input strategy now includes the newline character so Hypothesis can synthesise paragraph-shaped inputs. `lake build` accepts both predicates in ~ 0.2 s; the v4 predicate exercises more of Lean’s `stdlib` but stays well clear of `mathlib`.

Reference oracle (editable, v4). The same elicitation emits a Python oracle the reviewer can read off and rewrite if necessary. The v4 oracle (live) is a ~ 25 -line implementation that splits on `\n\n`, computes the first-line indent, flattens each paragraph’s lines into one word stream, and re-wraps:

```

def word_wrap(text: str, width: int) -> str:
    paragraphs = text.split('\n\n')
    result_paras = []
    for para in paragraphs:
        lines = para.split('\n')
        first_line = lines[0] if lines else ''
        indent = ''
        for char in first_line:
            if char == ' ': indent += char
            else: break
        words = []
        for line in lines:
            words.extend(line.strip().split())
        if not words:
            result_paras.append(''); continue
        wrapped_lines = []
        current_line = indent + words[0]
        for word in words[1:]:
            if len(current_line) + 1 + len(word) <= width:
                current_line += ' ' + word
            else:
                wrapped_lines.append(current_line); current_line = word
        wrapped_lines.append(current_line)
        result_paras.append('\n'.join(wrapped_lines))
    return '\n\n'.join(result_paras)

```

Section 5 of the iterative tab pre-fills this exactly text into a textarea with a yellow warning: “*oracle was LLM-synthesized — review (or rewrite) before trusting PBT verdicts.*” A reviewer who disagrees with, say, the empty-paragraph handling can rewrite that branch and re-run PBT without re-eliciting anything else.

Concrete test cases (live, exact output). “Generate cases” against the v4 spec produces 8 rows — one per edge condition the brief evolved to express:

1. `input=('hello world', 10)`
`expected='hello\nworld'` simple wrap at width 10
2. `input=(' hello world', 15)`
`expected=' hello world'` 2-space indent preserved
3. `input=(' hello world how are you', 15)`
`expected=' hello world\nhow are you'` indent on line 1 only
4. `input=('hello\nworld', 10)`
`expected='hello\nworld'` multi-line merged
5. `input=(' abc def\nghi jkl', 10)`
`expected=' abc def\nghi jkl'` indent + multi-line merge
6. `input=('para one\n\npara two', 15)`
`expected='para one\n\npara two'` blank line = new paragraph
7. `input='', 10)`
`expected=''` empty input
8. `input=(' a b c d ... v w x y z', 80)`

```
expected=' a b c d ... v w x y z' 4-space indent, fits
```

Rows 2, 3, 5, and 6 are exactly the four edge conditions the brief had to grow to express; they would not have appeared from the v1 spec. The reviewer’s job at this point is to confirm the expected values match what they actually meant (e.g. whether row 5’s expected is *really* ‘ abc def\nghi jkl’ and not ‘ abc\ndef ghi jkl’); cases the reviewer disagrees with are edited or deleted in place.

Why Example 1 matters. The headline is not the size or the depth of the v4 predicate — it is that *each round of refinement reused everything that did not change*. After editing v1’s brief into v2, the reviewer regenerated only the Lean and the cases; the starting repo, the existing test rows that still apply, and the export bundle layout were preserved. By v4 the spec bundle contained the brief’s full evolution, the final Lean predicate, the editable oracle, the eight reviewed test cases, and (if the reviewer chose to run it) the agent’s implementation with a PBT verdict — one JSON download, ready for code review, audit, or downstream tooling. This is what the iterative pipeline is for.

3.9.2 Example 2: clean one-shot intent — is_not_prime

Input. Brief: “Write a python function to identify non-prime numbers.” Starting repo: one stub file primecheck.py with `def is_not_prime(n): raise NotImplementedError`.

Drafted spec (live). Unlike Example 1, the v1 brief is already correct — there is no v2 to write. The elicitation produces structural invariants AND a behavioral block; the behavioral block carries the Lean predicate, the Python reference oracle, and the Hypothesis input strategy:

```
Structural:
OnlyFilesModified(['primecheck.py'])
NoNewImports(['os', 'subprocess', 'socket', 'requests',
              'urllib', 'http', 'shutil', 'ctypes', 'pickle'])
DiffSmallerThan(25)
NoSecretsInDiff()
PositiveTestPasses('test_is_not_prime.py')

Behavioral:
function_name : is_not_prime
signature : is_not_prime(n: int) -> bool
input_strategy: integers(min_value=0, max_value=200)
lean_predicate:
def isNotPrime (n : Nat) : Prop :=
  n < 2 ∨ exists k, 2 <= k /\ k < n /\ n % k = 0
python_oracle:
def is_not_prime(n: int) -> bool:
  if n < 2:
    return True
  return any(n % k == 0 for k in range(2, n))
```

Lean (Section 2 of the iterative tab). The emitted .lean carries both blocks (structural spec plus algorithmic isNotPrime). lake build ✓ in 0.21 s.

Generated code + structural verdict + PBT (Sections 3–5). The codegen produces an optimised sqrt-bound trial-division implementation:

```
def is_not_prime(n):
    if n < 2: return True
    if n == 2: return False
    if n % 2 == 0: return True
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return True
    return False
```

The verdict combines the structural validator with the PBT-against-oracle runner (live trace):

```

V OnlyFilesModified(['primecheck.py']) -- within scope
V NoNewImports(...) -- no new imports from forbidden set
V DiffSmallerThan(25) -- diff is 11 lines (within limit)
V NoSecretsInDiff() -- no credential-shaped strings
V PositiveTestPasses(test_is_not_prime.py) -- passed
V BehavioralSpecHolds -- no counterexample in 200 examples
                        vs reference oracle (0.32 s)
                        [count parsed from Hypothesis
                        --hypothesis-show-statistics]
Verdict: ACCEPT

```

Why Example 2 matters. It is the boring case: the brief was complete on the first try, every artifact verified, no edits were needed. The same pipeline that demanded four rounds for word-wrap produced a verdict in one round here. The iterative shape doesn’t penalise easy intents — the reviewer just skips the editing and clicks through. Two implementations agreeing — the agent’s sqrt-bound trial division and the elicited oracle’s naive $\text{any}(n \% k == 0 \text{ for } k \text{ in range}(2, n))$ — on 200 random inputs is a sharper claim than “passes the spec’s positive test”; the behavioral spec captured the *mathematical content* $(n < 2 \vee \exists k, \dots)$ which a reviewer can read off the emitted Lean independently of either Python implementation. If the agent had returned $n \% 2 == 0$ instead, Hypothesis would have shrunk to $n=9$ (or similar) and reported it.

(A shortened multi-file Dijkstra example appears in Appendix A.2.)

4 Experiments

4.1 Setup and datasets

Hand-authored corpus (60 pairs). 15 tasks (10 toy + 5 mutation-generated) each with four ground-truth-labelled candidates: CORRECT, OBVIOUS_WRONG, SUBTLE_WRONG, SCOPE_CREEP. Used for the validator-vs-baselines comparison, the per-invariant ablation, and the mutation harness.

Complex multi-file tasks (12 pairs). Three additional hand-authored tasks introducing realistic engineering scope: a Flask JWT middleware, a SQL schema migration with model dataclass update, and a per-IP rate limiter middleware.

External benchmark samples (25 problems). 5 problems each from MBPP, HumanEval, Big-CodeBench, HumanEval Pro, and LiveCodeBench, adapted into the AmbiguousBrief shape. JSONL fixtures are committed to the repo.

4.2 Validator vs. baselines

We compare five evaluators on the 60-pair corpus (Table 3).

Table 3: Validator vs. baselines on the 60-pair extended corpus. “sec / $\Delta\%$ FAR” is wall-clock seconds per percentage-point of FAR reduction relative to `positive_only`.

Evaluator	Acc	FAR	FRR	κ	Disc. power	sec/ $\Delta\%$ FAR
structured (ours)	98.3%	2.2%	0.0%	0.957	97.8%	31.9
positive_only	50.0%	66.7%	0.0%	0.200	33.3%	(base)
llm_judge	100%	0%	0%	1.000	100%	30.8
n12postcond [8]	75.0%	0%	100%	0.000	0%	30.8
prd_style_judge [9]	100%	0%	0%	1.000	—	30.8

structured reaches LLM-judge-comparable accuracy while making **zero API calls** during evaluation; the LLM judges burn 30–300 seconds per corpus run. Cohen’s κ of 0.957 quantifies discriminative power: a coin-flip would yield 0.

4.3 Per-invariant ablation

We drop one invariant from every spec in the corpus, re-measure FAR (Table 4).

Table 4: Per-invariant ablation. “ Δ FAR” is the new minus baseline (3.3%) FAR. Scope discipline and import blocking carry most of the safety; FilesUnchanged is dead weight on this corpus.

Invariant ablated	FAR with	FAR w/o	Δ FAR	Newly admitted
OnlyFilesModified	3.3%	26.7%	+23.3%	7
NoNewImports	3.3%	23.3%	+20.0%	6
DiffSmallerThan	3.3%	13.3%	+10.0%	3
NoSecretsInDiff	3.3%	6.7%	+3.3%	1
FilesUnchanged	3.3%	3.3%	0.0%	0

4.4 Mutation harness

Table 5: Mutation classification on the 60-pair corpus (179 total mutations). Zero brittle mutations is the key honesty signal: the spec’s invariants are doing work without being on the edge of false rejection on CORRECT candidates.

Kind	load_bearing	brittle	invisible
drop_invariant	27	0	33
drop_test	15	0	0
widen_scope	12	0	2
shrink_set	11	0	49
weaken_bound	6	0	24
Total	71 (39.7%)	0	108

4.5 Cross-dataset pipeline run

Table 6 shows the 4-step pipeline run on 25 problems sampled from five benchmarks spanning 2021–2025.

Table 6: 4-step pipeline run on 5 problems per dataset (25 total). “Codegen (initial)” is the first run; the bold column is after three targeted fixes described in §4.6.

Dataset	Year	Drafted	Lean ✓	Codegen ✓	Codegen (initial)
MBPP [11]	2021	5/5	5/5	4/5 (80%)	2/5 (40%)
HumanEval [12]	2021	5/5	5/5	5/5 (100%)	5/5 (100%)
BigCodeBench [13]	2024	5/5	5/5	4/5 (80%)	2/5 (40%)
HumanEval Pro [14]	2025	5/5	5/5	5/5 (100%)	1/5 (20%)
LiveCodeBench [15]	2024–25	5/5	5/5	5/5 (100%)	0/5 (0%)
Total		25/25	25/25	23/25 (92%)	10/25 (40%)

4.6 The LiveCodeBench fix

The initial 40% pass rate (LCB at 0%) revealed three avoidable failure modes that the pipeline’s per-invariant trace made precisely diagnosable:

1. The LLM-invented positive test did not match the benchmark’s canonical I/O contract. On LCB this was catastrophic — contests are stdin/stdout-shaped and the LLM’s invented assertions used different I/O conventions.
2. The function stub did not communicate the I/O contract. “Implement solve” was ambiguous; some models wrote functions that took a list of strings, others called `input()/print()` directly.
3. The codegen response parser was too strict to handle `<answer>`-tag-wrapped JSON output — several models prefaced the JSON with reasoning text.

Three targeted fixes brought the combined score to **92%**, with LCB at **100%**:

1. **Pre-author the positive test from each benchmark’s canonical tests.** LCB ships `public_test_cases` with the exact contest I/O; we parse it into a real pytest and pass it as a new `override_positive_test=` parameter to `draft_spec`. The LLM still drafts the structural invariants but the test is the contest’s own — so the codegen step is graded against the same oracle the benchmark uses.
2. **Explicit contract in the stub.** The LCB stub now says verbatim: “Convention: read EVERYTHING from the `stdin` string argument, return EVERYTHING that would be printed as a single string. Do NOT call `input()` or `print()`.”
3. **Lenient JSON extractor + stricter prompt.** A new `_extract_json_object` tries the whole text → strips code fences → strips `<answer>/<result>/<output>` tags → brace-counts the first `{...}` substring. The system prompt additionally forbids preamble and tags up front.

The `override_positive_test` hook is dataset-agnostic; any benchmark that ships canonical tests can pre-author them in one line of adapter code. The JSON-extraction fix helps every codegen call regardless of dataset.

4.7 Cross-model comparison example

On the under-specified brief “Add a `subtract(a,b)` function to `calculator.py`,” drafting with both Claude Sonnet 4.5 and Claude Haiku 4.5 yields (Table 7).

Table 7: Cross-model agreement on an under-specified brief. The disagreement on `forbidden_imports` is the spec-level signal that the input under-specifies what defenses the spec should erect.

Field	Sonnet	Haiku	Agreement
<code>allowed_files</code>	<code>[calculator.py]</code>	<code>[calculator.py]</code>	agree
<code>forbidden_imports</code>	9 modules	0 modules	disagree
<code>max_diff_lines</code>	10	10	agree
<code>check_secrets</code>	true	true	agree

5 Discussion

Iteration is the missing affordance, not just polish. Watching reviewers use the iterative tab makes one observation routine: the brief they start with is never the brief they end with. The word-wrap example evolved across four rounds; even the simpler `is_not_prime` example benefits from the editable oracle (a reviewer who is suspicious of the LLM’s choice of `any(n % k == 0 for k in range(2, n))` can rewrite it to `any(n % k == 0 for k in range(2, int(n**0.5)+1))` and re-run PBT in seconds). Locking these artifacts behind a one-shot generate-and-ship interface — as Kiro, TiCoder, and the batch four-step pipeline do — is what makes spec authoring feel like fighting the tooling.

Lean emission is 100% reliable; codegen is not. Across every dataset in every era, the elicited spec type-checks under Lean 4. This is exactly the Lean Atlas blind spot [3] — structural well-formedness does not catch semantic under-specification. The codegen round-trip is what surfaces semantic gaps: when the LLM-generated implementation fails the spec’s own positive test (or its concrete cases, or its PBT-against-oracle), the per-invariant trace tells the reviewer *which* layer was inadequate and how.

The improvement strategy generalises. The `override_positive_test` hook is dataset-agnostic. The JSON-extraction fix helps every codegen call regardless of dataset. The lesson — *do not delegate the oracle to the model; pre-author it when the dataset provides one* — is general.

Provenance chips are the missing review interface. Of our 25 cross-dataset runs, every drafted spec contains at least one default-grounded invariant the LLM filled in without evidence. Surfacing those chips in the elicitation UI gives reviewers the “needs semantic review” signal Lean Atlas argues a bare `lake build ✓` does not provide.

Mutation testing of specs (not code) is novel and cheap. Our harness runs 179 mutations on the 60-pair corpus in ~ 10 seconds. Per-spec coverage scoring is a Dodds-shaped honesty metric: a slide-deck-quality spec scores 0%, a defensible spec scores $\geq 60\%$.

6 Limitations

- The mutation harness’s `widen_scope` uses candidate-derived paths to remain informative; a blind variant would need curated scope-creep-shaped filename pools per category.
- All mutations are structural; semantic perturbation (e.g. inverting an assertion) is unimplemented.
- The complex hand-authored corpus is 3 multi-file tasks (12 pairs) — small relative to FeatureBench’s 200 tasks.
- `lake build` verifies logical, not semantic, correctness of emitted Lean. `looksLikeSecret` is opaque because regex matching on `String` is heavy to model.
- External-dataset adapters are shallow for benchmarks without canonical tests. Only Live-CodeBench currently gets the `override_positive_test` treatment; MBPP and HumanEval ship test cases we use as `existing_tests` sources but not as the oracle.
- Sample size on the 5-dataset run is 5 problems each (25 total). Headline numbers are best read as proof-of-concept, not a benchmark result.

7 Conclusion

Specifications for real systems emerge through iteration, not as one-shot artifacts. The right tooling shape for AI coding agents is therefore one in which the reviewer can edit any one artifact — Lean predicate, generated code, reference oracle, concrete test cases — and re-verify just that artifact without redoing the rest, then export the whole session as a single JSON spec bundle. We present an iterative pipeline of this shape, alongside a batch four-step sibling that uses the same modules for benchmarking. On a 25-problem cross-dataset sample spanning 2021 $\rightarrow$ 2025 the batch pipeline reaches 100% Lean type-check and 92% codegen-validation after a small number of targeted, auditable fixes; the iterative pipeline absorbs four-round brief refinement on a word-wrap task we trace end-to-end. A deliberately partial structural-invariant DSL underlies both, drafted from ambiguous English in seconds, emitted as real Lean 4 type-checkable in 0.2 s, mutation-tested to surface which invariants are doing real work, and cross-validated against a second LLM to flag under-specified fields.

Future work: (i) a Dafny / F* back-end alongside Lean to compare formal-target ergonomics; (ii) a larger-scale FeatureBench / PRDBench run once access becomes available; (iii) semantic mutation operators (assertion inversion, parameter swap) to complement the current structural mutation harness; (iv) integration with a real PR-review surface (GitHub Checks) so the per-invariant trace lands in the agent’s actual workflow.

Reproducibility

All code, the Lean prelude, the evaluation harness, and the browser demo are in the bundled repository (<https://github.com/linhlt-it-ee/Hackathon-Vibe-Coding>). Reproducing the numbers in this paper:

```
# 4.2 + 4.3 (validator vs baselines, per-invariant ablation)
PYTHONPATH=. python3 -m safe_scaffold.cli task-eval --extended --rigorous --ablation

# 4.4 (mutation harness on the 60-pair corpus)
PYTHONPATH=. python3 -m safe_scaffold.cli mutate
```

```
# 4.5 + 4.6 (cross-dataset 4-step pipeline)
PATH="$HOME/.elan/bin:$PATH" PYTHONPATH=. python3 -m safe_scaffold.cli \
  dataset-run --dataset all --n 25 --no-compare

# Browser demo (everything together, all four steps live)
PYTHONPATH=. python3 demo_server.py # http://127.0.0.1:8765
```

The Lean toolchain is required for the type-check column:

```
curl https://raw.githubusercontent.com/leanprover/elan/master/elan-init.sh \
  -sSf | sh -s -- -y
```

ANTHROPIC_API_KEY is required for the LLM judges in Section 4 and for the elicitation / codegen calls in Section 4.5. 219 unit tests pass under `stdlib unittest`.

References

- [1] Dodds, M. (2025) Specifications Don’t Exist. Galois technical essay. <https://www.galois.com/articles/specifications-dont-exist>
- [2] Liu, S., Sridhar, N., Lampropoulos, L., et al. (2024) Daedalus: Safer Document Parsing. *Proc. ACM PL* (PLDI 2024). DOI: 10.1145/3656410.
- [3] Lin, J. et al. (2026) Lean Atlas: An Integrated Proof Environment for Scalable Human-AI Collaborative Formalization. *arXiv:2604.16347*.
- [4] Amazon Web Services (2026) Kiro: Spec-Driven Development IDE. <https://kiro.dev/docs/specs/>.
- [5] Sridharan, M. et al. (2014) SPEEDY: An Eclipse-based IDE for invariant inference. *arXiv:1404.6605*.
- [6] Wang, C. et al. (2023) Trustworthy Formal Natural Language Specifications. *Proc. ACM PL* (PLDI 2023). *arXiv:2310.03885*.
- [7] Lahiri, S. et al. (2022) TiCoder: Discriminating-Test-based Specification Elicitation. *arXiv:2208.05950*.
- [8] Endres, M. et al. (2024) Can Large Language Models Write Good Property-Based Tests? (nl2postcond). *arXiv:2310.01831*.
- [9] Fu, Y. et al. (2026) PRDBench / PRDJudge: PRD-graded evaluation of code agents. *AAMAS 2026 / arXiv:2510.24358*.
- [10] Zhou, M. et al. (2026) FeatureBench: Feature-Level Coding Tasks for LLM Evaluation. *ICLR 2026 / arXiv:2602.10975*.
- [11] Austin, J. et al. (2021) Program Synthesis with Large Language Models. *arXiv:2108.07732* (MBPP dataset).
- [12] Chen, M. et al. (2021) Evaluating Large Language Models Trained on Code. *arXiv:2107.03374* (HumanEval).
- [13] Zhuo, T.Y. et al. (2024) BigCodeBench: The Next Generation of HumanEval. *NeurIPS 2024 D&B Track*. <https://huggingface.co/datasets/bigcode/bigcodebench>.
- [14] Yu, Z. et al. (2025) HumanEval Pro and MBPP Pro: Evaluating LLMs on Self-Invoking Code Generation. *Findings of ACL 2025 / arXiv:2412.21199*.
- [15] Jain, N. et al. (2025) LiveCodeBench: Holistic and Contamination-Free Evaluation of Large Language Models for Code. *ICLR 2025 / arXiv:2403.07974*.

A Technical appendix

A.1 The browser demo at a glance

The demo exposes two top-level tabs that share the same backend modules. The *Iterative pipeline* tab (the primary contribution; Section 3.2) presents the five sections described in the methodology as collapsible <details> blocks: input + starting repo (the editor seeds with an empty `main.py` on first load), editable Lean with a `lake build` button, editable per-file Python with an `ast.parse` button, an editable test-case table with “Generate cases” and “Run against current code”

buttons, and an editable Python reference oracle with “Run PBT vs oracle” and “Clear oracle (skip PBT)” buttons. A single “Export bundle” button serialises the entire session state to JSON. The *4-step pipeline* tab is the batch sibling (Section 3.3): four full-screen sub-tabs with status badges, sequenced by a single “Run all 4 steps” button. Section 1 of the batch tab renders provenance chips per drafted invariant (DaeDaLus + Lean Atlas inspiration), a SPEEDY-style split-pane linking source phrases to the invariants they grounded, a Lean-Atlas-style mini dependency graph of intent $\rightarrow$ invariants $\rightarrow$ files, and the algorithmic Lean predicate + Python reference oracle from the `behavioral_spec` block. Section 2 toggles between `spec.lean` and a Kiro-style `requirements.ears` controlled-NL view. Section 3 shows the LLM-generated files with no verdict yet — the reviewer reads the code first. Section 4 shows the per-invariant trace plus the `BehavioralSpecHolds` row from PBT (verified, with the actual example count parsed from Hypothesis stdout; falsified, with the shrunk counterexample). The cross-model spec comparison lives in a separate “Compare drafts” top-level tab.

A.2 Multi-file worked example: vague Dijkstra fix

This example shows the same pipeline handling a multi-file refactor under a deliberately vague brief. We include it for completeness; the main-text walkthrough is Section 3.9.1.

Input. Brief: “*Fix `shortest_paths` in `pathfind.py`. We think there’s a bug in how it handles disconnected components. Also it’s $O(V^2)$ and we need $O((V + E) \log V)$ for the prod graph. One of the existing tests is wrong too — fix it. Don’t break the rest. Audit is next week.*” The brief is underspecified along several axes: it does not name *which* test is wrong, “don’t break the rest” is the only acceptance criterion, the complexity bound is named but the data-structure choice is not, and the scope unclearly straddles two files (`pathfind.py` and the test file). The starting repo contains a buggy Dijkstra (list-scan $O(V)$ per iteration; zero-default initialisation that makes unreachable nodes look reachable) plus three tests of which one asserts the buggy behaviour. The pipeline drafts a spec that correctly infers the two-file scope; codegen emits a heap-based Dijkstra with `math.inf` for unreachable nodes and rewrites the bad test to assert `math.inf`; the structural validator’s per-invariant trace verifies the diff stayed within the two-file scope (25 lines of 60), introduced no forbidden imports (`heapq` and `math` are allowed), and the rewritten positive test passes. Cross-model re-drafting flags three fields where Sonnet and Haiku disagree (`forbidden_imports`, `max_diff_lines`, `positive_test_loc`), directly signalling that the brief is muddier than a well-scoped one — exactly what the cross-model technique is designed for. The result is the Dodds-shaped outcome: *partial, honest, machine-checkable* — not *complete and coherent*, which the input never was.

A.3 The 3 hand-crafted ambiguous briefs

1. **Brief A — “do auth right.”** Extreme under-specification: one vague sentence about “tightening up” a Flask login flow, no PRD, no acceptance criteria. The LLM has to invent the scope.
2. **Brief B — “PRD vs itself.”** A multi-paragraph PRD that simultaneously requires bcrypt-hashed passwords, “store exactly as provided so support can read them back,” “no new dependencies,” and “use bcrypt.”
3. **Brief C — “slides vs tests.”** Design slides assert negative percent = surcharge and full discount = item is free plus \$5 credit; existing tests in the repo assert the opposite.

NeurIPS Paper Checklist

1. Claims

Question: Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope?

Answer: [Yes]

Justification: The abstract lists six concrete numbers (98.3% accuracy, 2.2% FAR, $\kappa=0.957$, 39.7% load-bearing mutations, 100% Lean type-check, 92% codegen-validation) each of which is reproduced verbatim in Section 4 from the actual commands in the Reproducibility section. The introduction’s six contributions map one-to-one to subsections of Section 3.

2. Limitations

Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: Section 6 enumerates six concrete limitations: candidate-derived `widen_scope`, no semantic mutations, small complex-task corpus (12 pairs), `lake build` catching only logical not semantic correctness, shallow external-dataset adapters, and a 25-problem cross-dataset sample.

3. Theory assumptions and proofs

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper presents an empirical pipeline and engineering measurements; it makes no formal theoretical claims requiring proof. The Lean-emission section ships type-checked Lean source as artifact-level evidence, not as a theorem.

4. Experimental result reproducibility

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The Reproducibility section in the main paper gives the four exact CLI commands that produce every number in Tables 3–6. The bundled corpus, dataset adapters, Lean prelude, and validator are all in the repository at the URL on the title page.

5. Open access to data and code

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: All code, the Lean 4 prelude, the 60-pair hand-authored corpus, the 25-problem cross-dataset JSONL fixtures, the demo server, and 219 unit tests are released at <https://github.com/LidaSafety/vibespecs>. Reproduction commands and the optional Lean toolchain install are documented in the Reproducibility section.

6. Experimental setting/details

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Section 4.1 describes the three corpora (60-pair hand-authored, 12-pair complex, 25-problem cross-dataset). Section 4 lists every evaluator and its configuration; the LLM models used (Claude Sonnet 4.5, Haiku 4.5) are named in the cross-model example (Table 7). No training is performed; the validator is deterministic and uses no hyperparameters beyond per-invariant bounds (which are spec-authored, not tuned).

7. Experiment statistical significance

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [No]

Justification: We report Cohen’s κ and discriminative power on the deterministic validator runs (no variance to bar). For the LLM-dependent experiments (cross-dataset run in Section 4.5, cross-model comparison in Table 7) sample sizes are small (5 problems per dataset, 1 brief in the comparison example) and we report exact counts rather than confidence intervals. The paper acknowledges this in Section 6 (“Sample size on the 5-dataset run is 5 problems each (25 total). Headline numbers are best read as proof-of-concept”).

8. Experiments compute resources

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: The deterministic validator runs the 60-pair corpus in ~ 1.6 s on a stock laptop (Section 3.1); the mutation harness runs 179 mutations in ~ 10 s (Section 5); each Lean type-check takes ~ 0.2 s after the prelude is cached (Section 3.4); each LLM call costs 5–30s wall-clock and one Anthropic API call. No GPU is needed.

9. Code of ethics

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

Answer: [Yes]

Justification: The work uses publicly-released code-evaluation benchmarks under their published licenses; no human subjects are involved; the elicitation pipeline targets safer coding-agent behavior, which aligns with the Code of Ethics’ emphasis on responsible release.

10. Broader impacts

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: The work’s positive impact is in reducing scope-creep and forbidden-import classes of AI coding-agent failures (the introduction quantifies this on the corpus). The main potential negative impact is that a partial-spec safety net could be misread as a full proof of correctness; Section 5 and Section 6 explicitly call out that `lake build` verifies only logical, not semantic, correctness, and that the codegen round-trip is what catches semantic gaps. This honest framing is itself a mitigation.

11. Safeguards

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The released assets are a Python invariant DSL, a Lean prelude, evaluation code, and 25 adapter fixtures pointing at already-public benchmarks. No new models or scraped datasets are released; misuse risk is low.

12. Licenses for existing assets

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All five external benchmarks (MBPP, HumanEval, BigCodeBench, HumanEval Pro, LiveCodeBench) are cited in Section 2 with arXiv IDs and venue. Our adapters in `safe_scaffold/task_spec/datasets/` bundle 5 samples per benchmark under each project's published license; the adapter source files credit the original authors in their docstrings. The NeurIPS 2026 style file (`neurips_2026.sty`) is the official template.

13. New assets

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [\[Yes\]](#)

Justification: New assets (the invariant DSL, the Lean prelude, the mutation harness, the demo server, the 3 ambiguous-input briefs, the 3 complex multi-file tasks) are all released with module-level docstrings, an `INSTALL.md`, the `docs/elicitation_and_mutation.md` writeup, and 219 unit tests in the repository.

14. Crowdsourcing and research with human subjects

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [\[N/A\]](#)

Justification: No crowdsourcing or human-subjects research was conducted.

15. Institutional review board (IRB) approvals or equivalent for research with human subjects

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [\[N/A\]](#)

Justification: No human-subjects research was conducted; IRB approval is not applicable.

16. Declaration of LLM usage

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research?

Answer: [\[Yes\]](#)

Justification: LLMs are a core, non-standard methodological component. Section 3.2 describes the four-step pipeline's reliance on LLMs in Step 1 (drafting the spec), Step 3 (cross-model comparison), and Step 4 (code generation); the specific models used are Claude Sonnet 4.5 and Claude Haiku 4.5 (Anthropic). All LLM-produced outputs are constrained to a structured JSON schema with downstream structural validation, and the deterministic Python validator is the final oracle in Step 4; no LLM was used as the source-of-truth grader for our headline numbers.