VERIFIED BUT VACUOUS: THREE FAILURE SURFACES OF LLM-DRIVEN VERIFIED CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

What should "verified" mean when evaluating LLM-generated code? Many existing pipelines report a single verification number (for example, `dafny verify` accepted N% of samples). We argue this collapses three properties that are logically nested and empirically separable: verifier acceptance (Surface A), *syntactic specification binding* (Surface B: does the proof obligation bind to the spec?), and *semantic specification fidelity* (Surface C: does the spec capture the informal requirement?). We contribute this three-surface decomposition plus the *ensures-binding gate*, a syntactic Surface-B instance for Dafny pipelines that emit distinguishable specification and implementation artifacts (static AST walk, $<$10 ms per component). Two empirical demonstrations show the surfaces separate. **First ($A \neq B$), via Li@Opus (a matched-model reproduction of Li et al. (2025)'s Dafny-as-intermediate-representation (IR) pipeline under Claude Opus 4.7) on three Python benchmarks of simple coding tasks (LeetCode-style)**: it reaches **99.1% verifier acceptance** on HumanEval while only **1.9%** carry a certificate binding to a spec-defined predicate or function. This gap is **roughly fortyfold** (36 to 52$\times$) that holds across HumanEval, LeetCodeDataset, and LiveCodeBench v6 in a narrow **1.9% to 2.7% band**. This gap is observed even as the fraction of verified samples that carry *any* `ensures` clause climbs from 5.3% on HumanEval to 10.9% on LiveCodeBench v6. Postcondition prevalence more than doubles with benchmark difficulty but binding-to-a-spec-predicate rate does not follow. **Second ($B \neq C$), via VCG on two Rust benchmarks**: we use VCG (which is our own in-house Dafny-as-intermediate pipeline) on Rust because the Rust target lets us enforce fully-verifiable output (every component ships as a Dafny certificate, ensures-binding gate passing by construction). Even inside that construction, **11.4% (MBPP-Rust) and 14.82% (VeriContest LC slice) of structurally-verified samples still fail hidden tests**. On the LC slice of VeriContest, **VCG's End2End rate is 7.79% (535/6870) at pass@1, +2.3 pp above the best previously published End2End LC number (GPT-5.5's 5.51%) and 4.14$\times$ the matched-backend single-shot Opus 4.7 End2End LC baseline (1.88%)** that the VeriContest authors report per-slice. Therefore, any pipeline shipping certificates should report at least one of Surface B (e.g. the ensures-binding gate) or Surface C (e.g. Post2Exe) alongside Surface A; where feasible, both, since Surface B alone leaves the semantic blind spot we demonstrate on VCG, and Surface C alone leaves the syntactic blind spot we demonstrate on Li@Opus. Many existing evaluations report neither.

1 INTRODUCTION

What should "verified" mean when evaluating LLM-generated code? Large language models can generate code that passes tests most of the time. Frontier models solve 70–95% of HumanEval-style problems (Chen et al., 2021) and 60–90% of LeetCode-style benchmarks (newfacade, 2024; Jain et al., 2024). Passing a test suite tells us that no *tested* input triggered a bug; it does not tell us that the implementation satisfies the requirements on inputs the tests did not visit. For code shipped into production (payment processors, medical dose calculators, security-sensitive parsers), that distinction is crucial.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Formal verification systems (Dafny (Leino, 2010), Lean (de Moura & Ullrich, 2021), F* (Swamy et al., 2016), Verus (Lattuada et al., 2024)) can machine-check that an implementation satisfies its formal specification. Recent work has explored LLMs as authors of specs (Ma et al., 2024), impls (Misu et al., 2024), proof tactics (Yang et al., 2023), and complete verified programs (Xie et al., 2026; Li et al., 2025), with promising but basic-level empirical evidence. Existing evaluations of such pipelines typically report a single verification number alongside pass@k: `dafny verify` accepted N% of the samples, or `verus` returned zero on N% of them. This paper argues that a single verification number collapses three properties that are logically nested, but empirically separable. They are also independently non-trivial to satisfy.

We came to this decomposition while building **VCG**, an in-house Dafny-as-intermediate pipeline for LLM-driven verified code generation. VCG's pre-launch audit surfaced a class of `dafny verify`-accepted components whose `ensures` clauses did not reference any spec-defined predicate/function: the proof obligation was trivially provable against the impl body alone, and the certificate was vacuous *with respect to the intended functional requirement* (Dafny still established local static obligations, that is, well-formedness, termination, and absence of runtime errors, under whatever `expect`-style fixture assumptions were in scope; it just did not establish any general functional postcondition). We added a static ensures-binding integrity check to VCG's verify-repair loop to reject such components. That check turned out to be the *simplest instance* of a distinct evaluation surface we had been conflating with `dafny verify` acceptance.

Applying the same static check to a matched-model reproduction of Li et al. (2025)'s Dafny-as-intermediate pipeline (under Claude Opus 4.7) makes the case concretely: **`dafny verify` accepts 99.1% of samples on HumanEval, but only 1.9% carry a certificate that structurally binds its proof obligation to a spec-defined predicate/function.** Fully 94.7% ship no `ensures` clause at all; another 3.4% contain an `ensures` clause that does not bind to a spec-defined predicate/function. This is not a defect of Li et al.'s pipeline: their few-shot prompt (Figure 6 of Li et al. 2025, reproduced verbatim in our few-shot) is a bodyless `method {:testEntry}` with no postcondition, plus a `Main()` full of `expect` statements. Dafny *assumes* those expectations (not proves them) and checks them at runtime; on this input shape, verifier acceptance establishes the remaining static proof obligations (well-formedness, termination, local method-body proof) under those fixture assumptions, but does not prove the fixture expectations themselves or a general functional postcondition. The mismatch is a defect of the *metric* we use to summarize the pipeline, not of the pipeline itself.

Contributions. This paper isolates and develops the methodology contribution that motivated our own pipeline design:

1. **A decomposition of verification quality into three nested, empirically separable failure surfaces** (§2): Surface A (verifier acceptance), Surface B (*syntactic specification binding*), and Surface C (*semantic specification fidelity*). We contribute the **ensures-binding gate** (§3) as a syntactic instance of Surface B for Dafny pipelines that emit distinguishable specification and implementation artifacts: a static AST walk that runs in <10 ms per Dafny component.

2. **A three-level analytical decomposition of Surface B, and an empirical $A \to B$ gap of roughly fortyfold on Li@Opus** (§5). To avoid overclaiming what one syntactic gate establishes, we report certificate quality at three observable levels: **Level 1 =** verifier accepted (Surface A); **Level 2 =** `Impl.dfy` contains at least one `ensures` clause (architecture-agnostic postcondition presence, which does not require the clause to reference any spec-defined symbol); **Level 3 =** at least one `ensures` clause binds to a spec-defined predicate/function (our gate; a Surface-B instance for pipelines that separate spec and impl artifacts). On a matched-model reproduction of Li et al.'s Dafny-as-IR pipeline applied to three Python benchmarks of simple coding tasks (HumanEval, LeetCodeDataset, LiveCodeBench v6; LeetCode-style), Level 1 stays at 92–99% while Level 3 stays in a narrow **1.9–2.7% band** which has a **roughly fortyfold** (36–$52\times$) gap. The intermediate Level 2 climbs from 5.3% on HumanEval to 10.9% on LCB v6: the pipeline writes more postconditions on harder problems, but they do not get bound to spec-defined symbols. Level 2's growth without Level 3's growth is what makes the finding architecture-agnostic. It is not that our particular gate is too strict, it is that the additional postconditions at difficulty are inline rather than externalised. The narrow Level-3 band suggests the prompt/specification convention is

a stronger limiter of externalised-spec production than benchmark difficulty; whether it persists on further benchmarks is a testable prediction we leave open.

3. **Semantic-fidelity finding: Surface B → Surface C also leaks** (§6). Even inside VCG, which passes our external-spec-binding Surface-B criterion on 100% of shipped verified components by construction (other conceivable Surface B instances may still fail on this population), **11.4% (MBPP-Rust; Rust) and 14.82% (VeriContest LC; Rust) of structurally-verified samples still fail hidden tests**. Under the standard trusted-toolchain and correct-oracle assumptions, this operationalises spec fidelity as a distinct failure surface.

The sharper claim is that **verification quality must be decomposed**, and that many existing evaluations report only its weakest surface (AxDafny (Breen et al., 2026) and VeriContest's Post2Exe check (Xie et al., 2026, §4.1) are notable exceptions; Table 6 in Appendix D). §9 closes with the concrete methodological ask that follows: any pipeline shipping certificates should report at least one of Surface B or Surface C alongside its verifier-acceptance rate.

Figure 1 (Appendix A) visualises the cascade: Li@Opus's collapse at $A \to B$ vs. VCG's collapse at $B \to C$, with bar widths proportional to the fraction of attempted samples surviving each surface.

2 THREE FAILURE SURFACES OF LLM-DRIVEN VERIFIED CODE GENERATION

LLM-driven formal verification can fail at three logically distinct, empirically separable surfaces. They are nested: Surface B is evaluated on the population that passes Surface A; Surface C is evaluated on the population that passes Surface B. Success at one does not imply success at the next.

Surface A: Verifier acceptance. Did the verifier return zero on the emitted proof? For Dafny targets, `dafny verify` returning zero. For Verus targets, `verus` returning zero. Necessary but weakest, as we will show in this paper.

Surface B: Syntactic specification binding. Of the certificates the verifier accepts, what fraction actually bind the proof to something meaningful? We propose the **ensures-binding gate** (§3) as a concrete syntactic instance for Dafny: the method must contain at least one `ensures` clause, and at least one such clause must reference an identifier defined as a `predicate` or `function` in the paired specification artifact. This is a *syntactic, necessary-condition* heuristic. It catches certificates whose proof obligation is trivially provable against the impl body alone (the "vacuous" case), including the pathological subcase where the emitted spec contains no `ensures` clause at all. It does *not* catch semantically-vacuous predicates such as `predicate P(x: int) { true }`, nor does it establish that the predicate correctly captures the informal requirement; those failure modes live in Surface C.

Surface C: Semantic specification fidelity. Of the specs that survive Surface B, what fraction actually capture the *requirement* the user wanted? This is what VeriContest's Post2Exe check (Xie et al., 2026, §4.1) addresses: compile the emitted postcondition to an executable predicate and run it against positive and negative test cases. If the predicate admits outputs a negative test would reject, the spec is incomplete. Post2Exe is a *semantic* check complementary to our syntactic Surface B.

Concretely:

- **Surface A fails** when the verifier rejects the proof.
- **Surface B fails** when the verifier accepts a proof of the *wrong thing* because the spec is structurally trivial or absent (§5 measures this on Li@Opus, Python).
- **Surface C fails** when the verifier accepts a proof of a well-formed spec that nonetheless doesn't capture the intended behaviour (§6 measures this on VCG, Rust, which passes our external-spec-binding Surface-B criterion by construction).

Illustrative failure modes. Two minimal Dafny snippets on a canonical "return the maximum of `nums`" task make the separation concrete. **Surface A passes, Surface B fails**: the verifier accepts, but no ensures clause binds to the spec predicate.

Table 1: The three surfaces are nested and separable. Each surface's check is a static property of the emitted certificate; passing one level does not imply passing the next.

Surface	Check	Catches	Blind spot (deferred)
A: verifier acceptance	`dafny verify` (or `verus`) exits zero	proof gaps, type errors, missing lemmas	vacuous or absent `ensures` clauses
B: syntactic spec binding	some `ensures` clause references a spec-defined `predicate`/`function`	certificates that prove trivia unrelated to the spec artifact	semantically weak predicates that bind but under-specify
C: semantic spec fidelity	postcondition accepts positive tests *and* rejects negatives (Post2Exe)	weak specs whose predicates admit incorrect outputs	requirements no test suite happens to exercise

```
predicate MaxCorrect(nums: seq<int>, r: int)
  requires |nums| > 0
{ (forall i :: 0 <= i < |nums| ==> nums[i] <= r)
  && (exists i :: 0 <= i < |nums| && nums[i] == r) }

method find_max(nums: seq<int>) returns (r: int)
  requires |nums| > 0
{ r := 0; }                    // dafny verify exits 0 (no ensures)
```

The method has no postcondition, so the verifier has nothing to prove; the strong `MaxCorrect` predicate is left dangling. Surface A passes; the ensures-binding gate rejects.

Surface A and Surface B pass, Surface C fails: the ensures clause binds to a named spec predicate, but the predicate is too weak to constrain the implementation.

```
predicate MaxCorrect(nums: seq<int>, r: int)
  requires |nums| > 0
{ r >= 0 }                     // under-specified spec

method find_max(nums: seq<int>) returns (r: int)
  requires |nums| > 0
  ensures MaxCorrect(nums, r)  // syntactic binding satisfied
{ r := 0; }
```

The ensures clause references `MaxCorrect`, so the gate accepts. But `MaxCorrect` accepts any non-negative output; on input `[5,3,7]` the shipped method returns `0` and the postcondition still evaluates to true. Post2Exe (Surface C) catches this by running the postcondition against a negative test case for which $0 \geq 0$ falsely accepts a value the requirement rules out.

A pipeline shipping certificates ought to report at least one of Surface B or Surface C alongside Surface A. Many existing evaluations report only Surface A.

3 The ensures-binding gate: a syntactic instance of Surface B

Let M be a Dafny method with `ensures` $E_1, \ldots, E_n$, and let S be the paired specification (`Spec.dfy` plus any transitively-included support files). Define $\mathrm{Preds}(S)$ as the set of `predicate` and `function` symbols declared in S, and $\mathrm{Ids}(E_i)$ as the set of identifiers referenced in ensures clause E_i. The ensures-binding gate accepts M iff:

$$n \geq 1 \ \wedge \ \exists i, \exists p \in \mathrm{Preds}(S) \ : \ p \in \mathrm{Ids}(E_i)$$

Equivalently: the method has at least one `ensures` clause, and at least one of those clauses names an identifier that refers to a spec-defined predicate/function. The $n \geq 1$ conjunct is explicit so the condition is not vacuously satisfied by methods with zero postcondition clauses. The existential form (some clause binds, not every clause) matches the reference implementation (Appendix C) and

preserves certificates that pair one binding functional postcondition with auxiliary sanity postconditions that need not independently reference the named predicate. The check is a static AST walk, cheap to run (<10 ms per component in practice), and is drop-in for **Dafny pipelines that emit distinguishable specification and implementation artifacts** (the architectural regime this paper measures against).

What the gate catches. Vacuous certificates where the LLM-emitted `ensures` clause is trivially provable against the impl body alone without invoking any spec predicate. A canonical example (from our pre-launch audit of an earlier draft of our own pipeline):

```
predicate ComputeXCorrect(input: int, result: int)
  requires input >= 0
{ result >= 0 && result == input * 2 }

method compute_x(input: int) returns (result: int)
  requires input >= 0
  ensures result >= 0            // trivially provable, unbound to spec
{ result := 0; }                 // dafny verify returns zero
```

The `ensures result >= 0` clause is provable against `result := 0` without invoking `ComputeXCorrect`; the shipped code is wrong. The gate rejects this statically at zero SMT cost. It does *not* catch vacuous specs whose predicate itself does not capture the requirement (Surface C) nor vacuously-easy predicates like `predicate P(x: int) { true }`; those live in §6.

Empirical anchor. In VCG's pre-launch internal audit (across three Python benchmarks), **∼27% of components that had passed `dafny verify` failed this gate**. This is roughly one in four blessed certificates that was proving something trivial against a spec that did not name a predicate. This is the concrete frequency the gate is designed to catch inside a pipeline that is actively trying to write real specs. §5 shows that on a pipeline that is not trying to do that (i.e., Li et al.'s Dafny-as-intermediate approach reproduced on Claude Opus 4.7), the failure rate is not 27% but ∼98%.

4 Experimental setup

The two demonstrations in §5 and §6 share a common backend and evaluation protocol. This section consolidates the setup so those sections can report results without re-introducing it.

Benchmarks. Five benchmarks across two target languages (Table 2). The three Python benchmarks (HumanEval, LeetCodeDataset, LiveCodeBench v6) are LeetCode-style function-completion tasks and drive the $A \to B$ measurement in §5. The two Rust benchmarks (MBPP-Rust, VeriContest LC slice) drive the $B \to C$ measurement in §6, where the Rust target lets us enforce fully-verifiable output.

Table 2: Benchmarks used in this study. *Samples/prob.* is 10 across all VCG, BareLLM, and Li@Opus runs; the VeriContest authors' reported per-slice Opus 4.7 End2End LC number (1.88%; our matched-backend comparator in §6) is single-shot per problem.

Benchmark	# problems	Samples/prob.	Target	Source
HumanEval / HumanEval+	164	10	Python	(Chen et al., 2021; Liu et al., 2023)
LeetCodeDataset	228	10	Python	(newfacade, 2024)
LiveCodeBench v6	1,055	10	Python	(Jain et al., 2024)
MBPP-Rust	208	10	Rust	(Emergence AI, 2026)
VeriContest LC slice	687	10	Rust	(Xie et al., 2026)

Pipelines compared. Three pipelines share Claude Opus 4.7 as the sole backend. **BareLLM** is the matched-backend minimal baseline: Opus 4.7 emits target-language code directly with no spec, no proof, and no verification scaffolding (used for the pass-rate tax comparison in §7). **Li@Opus** is a matched-model reproduction of Li et al. (2025)'s Dafny-as-intermediate-representation pipeline; the

few-shot prompt is verbatim from Figure 6 of Li et al. (2025) (reproduction details in Appendix B). **VCG** is our own seven-agent pipeline: Planner decomposes the problem into components; SpecAgent authors Dafny specs; ImplementationAgent authors implementations; ProverAgent drives a verify-repair loop with `dafny verify` and the ensures-binding gate; verified components transpile to the target language via `dafny build -target:{py,rs}`. The two LC-slice *External* comparators we cite, the VeriContest authors' per-slice matched-backend Opus 4.7 End2End LC number (1.88%) and the strongest previously published End2End LC number (GPT-5.5's 5.51%), come directly from Xie et al. (2026) and are not re-run by us.

Model, sampling, and verifier. All LLM calls hit Anthropic Claude Opus 4.7 via Google Vertex AI. VCG's verifier stack is **Dafny 4.10 with Z3 4.13**. Every VCG-shipped verified component passes: (i) `dafny verify`; (ii) the ensures-binding gate (§3); (iii) an extracted-example gate on visible test cases from the problem description; and (iv) `dafny build` to the target language.

Statistical treatment. pass@1 and pass@10 are computed per Chen et al. (2021)'s unbiased estimator over the 10 samples/problem. Reported rates are point estimates; we do not report confidence intervals in the main body. The *cross-model* claim is limited to matched-backend statements (all three pipelines on Opus 4.7); we do not extrapolate to other backends. The VeriContest LC-slice End2End comparator (1.88%, single-shot per problem) has a different sampling budget than our 10-samples-per-problem VCG; the 4.14× ratio in §6 sits on that asymmetry (a real caveat, not a bug).

Reproducibility. Every run is described by a `manifest.json` recording the pipeline commit SHA, LLM model identifier and checkpoint, sampling settings, random seeds, benchmark version, and the exact CLI flags used. All tables and figures in this paper are reproducible from those per-run artifacts.

5 Verifier acceptance does not imply syntactic specification binding: the Li@Opus $A \to B$ gap

We apply the ensures-binding gate to every verifier-accepted sample from a matched-model reproduction of Li et al. (2025)'s Dafny-as-intermediate pipeline ("Li@Opus"; reproduction details in Appendix B). Certificate quality is reported at the three observable levels introduced in Contribution 2: Level 1 (verifier accepted; Surface A), Level 2 (any `ensures` clause; architecture-agnostic presence check, an inline `ensures r == 2 * x` counts), and Level 3 (an `ensures` clause binds to a spec-defined `predicate`/`function`; our gate). Level 3 is an external-spec-binding check for pipelines that separate spec and impl artifacts, not a universal test of postcondition meaning, as we will show in this paper.

Table 3: Certificate quality on Li@Opus, Claude Opus 4.7, three Python benchmarks. Level-2 and Level-3 rates are conditional on verifier acceptance (Surface A); raw `dafny verify` pass rates are 99.1% / 97.2% / 92.2%.

	HumanEval	LeetCodeDataset	LCB v6
Total samples (attempted)	1,640	2,280	10,550
Level 1: verifier-accepted (denom.)	1,625	2,217	9,731
Level 2: any `ensures` clause	5.3%	9.6%	10.9%
Level 3: ensures-binding gate passes	**1.9%**	**2.7%**	**2.3%**

Level 2 alone already exposes the gap, architecture-agnostically. Nine out of ten Li@Opus "verified" samples have no `ensures` clause anywhere in the generated `Impl.dfy` (94.7% / 90.4% / 89.1% respectively). The "specification" is a bodyless `method {:testEntry}` signature with no postcondition, plus a `Main()` full of `expect` statements that Dafny *assumes* (not proves) at verify time. What the verifier certifies is that these assumed fixtures are consistent enough to admit a local method-body proof. It is not functional correctness against a specification. This is faithful to Li et al.'s benchmarked pipeline, not a replication defect: Figure 6 of Li et al. (2025), reproduced verbatim in our few-shot prompt, has exactly this shape.

Level 3 is roughly fortyfold smaller than Level 1 (36–52× across the three benchmarks), and roughly benchmark-invariant. The gate-pass rate stays in a narrow **1.9–2.7% band** across all three benchmarks despite level-2 prevalence more than doubling from HumanEval (5.3%) to LCB v6 (10.9%) as benchmark difficulty rises. Additional `ensures` clauses at higher difficulty add inline postconditions not tied to a named symbol, rather than externalising specifications. The narrow range suggests the prompt/spec convention is a stronger limiter than benchmark difficulty; whether it persists on further benchmarks is a testable prediction. Other structural (Surface B) and semantic (Surface C) checks are possible (see §3, §8, Table 6 in Appendix D).

6 Syntactic binding does not imply semantic fidelity: the $B \to C$ leak in VCG

The $A \to B$ gap in §5 is stark because Li@Opus's pipeline puts essentially no structural pressure on the spec. What about a pipeline that *does* pass our external-spec-binding instance of Surface B by construction? Does that guarantee Surface C?

We measure this on our own architecture VCG, evaluated on two Rust benchmarks (MBPP-Rust and VeriContest LC). VCG is a seven-agent Dafny-as-intermediate pipeline that runs the ensures-binding gate inside the prover's verify-repair loop and downgrades any component whose certificate would fail the gate. Across five benchmarks and two target languages evaluated in this study, VCG ships **18,008 verified components, 100% of which pass the ensures-binding gate**, i.e. 100% pass our external-spec-binding Surface-B criterion (other conceivable Surface B instances may still fail on this population; we do not check those).

Yet some of those shipped-and-structurally-verified samples still fail hidden tests. This provides operational evidence that Surface C (semantic specification fidelity) is a distinct failure surface, under the trusted-toolchain and correct-oracle assumptions stated below.

Surface C operationalised on the two Rust benchmarks. Under the VCG Rust configuration, every component becomes a verification target and no unverified glue is shipped. Under the standard trusted-toolchain and correct-oracle assumptions (Dafny transpilation is sound; the Rust harness runs the intended oracle; the wrapper/marshalling layer preserves semantics), hidden-test failure among structurally-verified all-component samples operationalises Surface C: the certificate closed a proof that did not capture the requirement. A sample is *fully-verified* iff every component in its build package shipped as a gate-passing Dafny certificate.

Two Surface-C instances reported in this section. Both condition on samples that have already cleared Surfaces A and B, but they ask different questions and can catch different failures (Table 4). Table 5 reports the first (hidden-test failure) on both Rust benchmarks; the Post2Exe End2End @1 rate is discussed later in this section on VeriContest.

Table 4: Two Surface-C instances reported in this paper. Both condition on samples that have already cleared Surfaces A and B. They ask different questions and can catch different failure modes.

	(i) Hidden-test failure	**(ii) Post2Exe**
Question asked	Does the *implementation* match the requirement?	Does the *postcondition itself* match the requirement?
Mechanism	Run the shipped code against hidden tests	Translate the postcondition to an executable predicate; check it accepts positives and rejects negatives
Reported in	Table 5 (both Rust benchmarks)	End2End @1 (VeriContest LC)

The Rust rates on the first operationalisation are in Table 5.

Expressed as a share of ALL attempts (not conditional), the Rust residue is small: 5.9% of MBPP-Rust samples (124/2080) and 3.33% of VeriContest-LC samples (229/6870) are fully-verified but fail tests. These are the two Surface-C proxies reported in this paper; because hidden-test failure can in principle also reflect transpiler unsoundness, harness/oracle disagreement, or benchmark-oracle errors, spec inadequacy is the most likely but not the only contributor.

Table 5: VCG on the two Rust benchmarks. *Fully-verified* samples pass our Surface-B (ensures-binding) criterion by construction and ship a `dafny verify`-accepted certificate on every component. $B \to C$ *failure rate* is the fraction of those that nonetheless fail hidden tests, a Surface-C proxy under the trusted-toolchain and correct-oracle assumptions. *Verified-and-passing rate* is the same quantity expressed as a share of all attempts (10 samples per problem).

Benchmark (Rust)	Total attempts	Fully-verified	$B \to C$ fails	$B \to C$ failure rate
MBPP-Rust (Emergence AI, 2026)	2,080	1,080	124	**11.4%**
VeriContest LC (Xie et al., 2026)	6,870	1,545	229	**14.82%**

Why the leak exists even with the gate enforced. The ensures-binding gate is a syntactic necessary-condition check: it verifies that the certificate *mentions* the spec's predicates. It cannot verify that those predicates *capture the requirement*. A spec that names the right predicates but constrains them incompletely will still ship a structurally-gated certificate whose impl passes the proof but does not generalise to hidden tests. Closing that gap requires a semantic check (Post2Exe (Xie et al., 2026, §4.1) is one option; surface-grounded spec strengthening, lifting concrete examples from the problem description into formal lemmas the spec must satisfy, is another we are pursuing).

Post2Exe as Surface C, applied to VCG on the VeriContest LC slice. We ran Post2Exe (the semantic-fidelity check from Xie et al. 2026) on all 1,316 VCG-verified+passing samples on the VeriContest LC slice (687 problems, 10 samples/problem). **535 pass all four End2End checks (precond-equivalence, postcond-Post2Exe, CodeGen, ProofGen), giving an End2End @1 of 7.79% (535/6870). This is a new SOTA on the LC slice at matched-backend Opus 4.7: +2.3 pp above the best previously published End2End LC number (GPT-5.5's 5.51%) and 4.14× the matched-backend single-shot Opus 4.7 End2End LC baseline (1.88%)** that the VeriContest authors report per-slice. The Post2Exe keep-rate among verified-and-passing LC samples is 40.7% (535 / 1,316). CF-slice evaluation is left to future work.

7 Real specifications cost pass rate

Both VCG and Li@Opus show the same pattern in the runs measured here: samples that carry a non-trivial `ensures` clause pass tests less often than samples without one.

The effect is quantified in Table 7 (Appendix E): -14.0 pp on HumanEval ($n = 86$ vs. 1,539), -6.5 pp on LeetCodeDataset ($n = 213$ vs. 1,995), and -9.4 pp on LiveCodeBench v6 ($n = 1{,}032$ vs. 8,416); samples with any `ensures` clause vs. samples without, respectively.

In every case, writing a real postcondition costs pass rate. The mechanism is straightforward: a constraining spec cuts off certain LLM outputs the model would otherwise produce and which happen to pass the tests. This is the *same phenomenon* as VCG's pass tax against a bare-LLM baseline (matched-backend Opus 4.7 that emits code directly without a spec or proof; 3–5 pp pass-rate tax on Rust runs; larger on Python), at the architectural rather than per-sample scale, pipelines pay a functional-correctness cost for the discipline of writing constraining specifications.

The methodological consequence is that when a pipeline that externalises specifications appears *worse* on pass@k than a pipeline that does not, the difference is at least partly explained by which pipeline is doing more real spec-writing. Reporting pass@k alongside a Surface-B rate makes the tradeoff legible; reporting pass@k alone hides it.

8 Related work

LLM-driven verified code generation. The closest prior work is Li et al. (2025) (Dafny as intermediate language for LLM-driven verified Python) and VeriContest (Xie et al., 2026) (a 946-problem verifiable-Rust benchmark with a Post2Exe semantic check, our closest Surface-C analogue). AutoSpec (Wen et al., 2024) and Clover (Sun et al., 2023) investigate LLM-generated Dafny specs specifically. VerusBench (Yang et al., 2024), RepoVBench (Zhong et al., 2025), VeruSAGE (Yang et al., 2025), and VeriStruct (Sun et al., 2025) evaluate LLM proof generation in Verus at different

scales but do not evaluate end-to-end NL → verified code in the same shape as Li et al. (2025) or VeriContest (Xie et al., 2026).

Certificate integrity and vacuous verification. Prior work has noted that verifier acceptance is insufficient: Dafny's `{:axiom}` and `{:verify false}` attributes can bypass proof obligations entirely (Dafny Reference Manual, 2024, and community folklore), and any pipeline that ships specs must decide how to handle those attributes. Our paper generalises: the "obvious" vacuity of `{:axiom}` sits alongside the subtler class of *no `ensures` clause at all* (a certificate that proves what Dafny checks even without any postcondition), which the ensures-binding gate catches statically. AxDafny (Breen et al., 2026) independently converges on a related family of Surface-B-style checks (missing `ensures`, trivially-true postconditions, bypass constructs, plus an LLM reviewer for other cheating patterns), see Table 6 in Appendix D for a side-by-side positioning. Clover (Sun et al., 2023) proposes closed-loop consistency checks among code, spec, and tests as a related but semantic-rather-than-syntactic integrity mechanism. VeriContest's Post2Exe check (Xie et al., 2026, §4.1) is the closest prior Surface-C instance: compile the emitted postcondition to an executable predicate and run it against positive/negative tests. Ensures-binding and Post2Exe are complementary, syntactic and cheap vs. semantic and expensive; a pipeline shipping certificates ought to report at least one.

Pass@k-only evaluations. Codex (Chen et al., 2021), AlphaCode (Li et al., 2022), and self-consistency / reflection / multi-agent follow-ups (Wang et al., 2023; Shinn et al., 2023; Qian et al., 2024) established the pass@k evaluation methodology on HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and LeetCode-style benchmarks. Code-efficiency metrics (Huang et al., 2024) also lie on the functional-correctness axis. Our three-surface decomposition is additive to these, not a replacement.

Our contribution is a metric decomposition, not a new pipeline: the three-surface decomposition is formalism-agnostic; our ensures-binding gate is a Dafny-specific Surface-B instance; Post2Exe (Xie et al., 2026, §4.1) is a complementary semantic Surface-C instance in Verus. The community is independently converging on structural (Surface B) checks; this paper names the surface and reports the paired A-vs-B measurement on a pipeline that does not enforce a Surface B check itself. Table 6 (Appendix D) places the closest prior pipelines and evaluations on the three-surface hierarchy.

9 Conclusion, Limitations, and Methodological Ask

"Verified" is not a single property but a three-surface hierarchy. In our matched-model reproduction of Li et al.'s pipeline, 99.1% Dafny verifier acceptance on HumanEval corresponds to only 1.9% external-spec binding, a **roughly fortyfold** gap holding across three Python benchmarks of differing difficulty. On our own architecture that closes this gap by construction, an 11.4% (MBPP-Rust) / 14.82% (VeriContest LC) $B \to C$ gap remains. On the VeriContest LC slice VCG's End2End @1 is **7.79% (535/6870), a new SOTA at matched-backend Opus 4.7 (+2.3 pp over GPT-5.5's 5.51% and 4.14× the 1.88% Opus 4.7 LC baseline)**. **The methodological ask:** any pipeline shipping certificates should report at least one of Surface B (e.g. the ensures-binding gate) or Surface C (e.g. Post2Exe) alongside Surface A; where feasible, both, since Surface B alone leaves the semantic blind spot we demonstrate on VCG, and Surface C alone leaves the syntactic blind spot we demonstrate on Li@Opus.

Limitations. **Single model family:** all primary results use Claude Opus 4.7. The gap appears driven primarily by prompt convention (Li et al.'s few-shot is a bodyless `method {:testEntry}` whose `Main()` runs `expect` statements at runtime; the model reproduces that shape) rather than model capability, but this is not directly measured on other models. Cross-model generalisation (Sonnet 4.7, GPT-5.5, Gemini 3.1 Pro, open-source models) is open work. **One Surface-B instance:** the ensures-binding gate is one concrete Surface-B check, not the only one; it does not catch `predicate P(x: int) { true }` trivialities, `{:axiom}`-bypassed proofs, or Post2Exe-detectable semantic gaps (Xie et al., 2026, §4.1). AxDafny (Breen et al., 2026) converges on a related Surface-B check that would also work. The methodological ask is "report *some* Surface-B or Surface-C instance," not specifically this gate.

10 AI DISCLOSURE SECTION

In this work, we used generative AI tools for assisting with “implement methods, support qualitative and thematic data analysis". We have not used generative AI tools for “interpret results", and “Generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean and reformat dataset", are not applicable to this work. Additionally, we used generative AI tools for “create or modify scientific figures or images, create or edit software code, creation of artifacts, draft parts of a research paper, summarize or analyse existing literature, brainstorming, sourcing/searching for information, identify relevant literature, format references". We have reviewed all AI-assisted work. LLM-generated code was verified and tested for correctness by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Blair Breen, Alexander Letson, Begoña Requena Pozo, and Luca Sarra. AxDafny: Agentic verified code generation in Dafny. *arXiv preprint arXiv:2606.32007*, 2026. Validation checks include missing `ensures`, trivially-true postconditions, proof-bypass constructs, and an LLM reviewer for other cheating patterns.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dafny Reference Manual. Dafny documentation: `{:axiom}`, `{:verify false}`, `{:extern}` attributes. `https://dafny.org/latest/DafnyRef/DafnyRef`, 2024.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Conference on Automated Deduction (CADE-28)*, 2021.

Emergence AI. MBPP-Rust: Rust port of the MBPP dataset, 208 problems covering task ids 11–510, 2026. Released alongside this paper’s evaluation artifacts.

Dong Huang et al. EffiBench: Benchmarking the efficiency of automatically generated code. *arXiv preprint arXiv:2402.02037*, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2024.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16)*, 2010.

Yue Li, Georg Zetzsche, and Siva Somayyajula. Dafny as verification-aware intermediate language for code generation. In *Dafny Workshop, co-located with POPL*, 2025. arXiv:2501.06283.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, et al. Competition-level code generation with AlphaCode. *Science*, 2022.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. EvalPlus / HumanEval+.

Yifei Ma et al. Specification generation for neural programs via large language models. *arXiv preprint arXiv:2402.11897*, 2024.

Md. Rakib Hossain Misu et al. Towards AI-assisted synthesis of verified Dafny methods. *arXiv preprint arXiv:2402.00247*, 2024.

newfacade. LeetCodeDataset. `https://huggingface.co/datasets/newfacade/LeetCodeDataset`, 2024.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. In *Association for Computational Linguistics (ACL)*, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Chuyue Sun, Yifan Sun, Dorsa Amrollahi, Eric Zhang, Shuvendu Lahiri, Shan Lu, David Dill, and Clark Barrett. VeriStruct: AI-assisted automated verification of data-structure modules in Verus. *arXiv preprint arXiv:2510.25015*, 2025.

Chuyue Sun et al. Clover: Closed-loop verifiable code generation. *arXiv preprint arXiv:2310.17807*, 2023.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages (POPL)*, 2016.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Cheng Wen et al. Enchanting program specification synthesis by large language models using static analysis and program verification. *arXiv preprint arXiv:2404.00762*, 2024.

Zhiyuan Xie, Mridul Pawagi, Yifei Liu, Ashutosh Rai, Lianmin Shao, Jason Berberian Jr., Sicheng Che, and Wenting Wang. VeriContest: A competitive-programming benchmark for verifiable code generation. *arXiv preprint arXiv:2605.08553*, 2026.

Chenyuan Yang, Arina Gladshtein, et al. AutoVerus: Automated proof generation for rust code. *arXiv preprint arXiv:2409.13082*, 2024. Introduces VerusBench.

Chenyuan Yang, Norrie Neamtu, Chris Hawblitzel, Jacob R. Lorch, and Shuvendu Lu. VeruSAGE: A study of agent-based verification for Rust systems. *arXiv preprint arXiv:2512.18436*, 2025. Introduces VeruSAGE-Bench.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Sicheng Zhong, Jiaqi Zhu, Yifan Tian, and Xujie Si. RAG-Verus: Repository-level program verification with LLMs using retrieval augmented generation. *arXiv preprint arXiv:2502.05344*, 2025. Introduces RepoVBench.

A CASCADE FIGURE: $A \to B$ VS. $B \to C$ COLLAPSE

Figure 1 shows the two-pipeline cascade referenced from §1.

B LI@OPUS REPRODUCTION DETAILS

Codebase. The reproduction consists of prompt files (verbatim from Figure 6 of Li et al. 2025), the verify-repair loop implementation, the compile/shim scripts, and per-benchmark harness adapters.

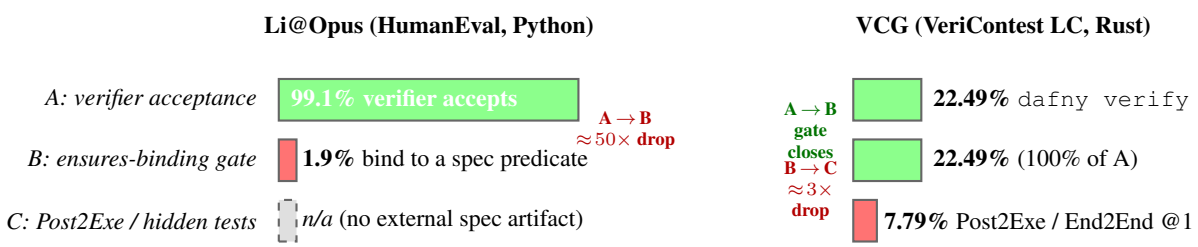


Figure 1: **Two pipelines fall out of the cascade at different surfaces.** Bar widths are proportional to the fraction of *attempted samples* surviving each surface (HumanEval: 1640 attempts; VeriContest LC: 6870 attempts). Li@Opus's collapse is at $A \to B$ (99.1% verifier acceptance, but only 1.9% carry a certificate binding to a spec-defined predicate). VCG closes B on 100% of A by construction; its collapse is at $B \to C$ (22.49% `dafny verify` passing to 7.79% End2End @1). A pipeline reporting only Surface A would show 99.1% and 22.49% respectively for Li@Opus and VCG and hide the shape of failure in both cases.

Adaptations per benchmark.

- **HumanEval / HumanEval+**: no adaptation; Li et al.'s prompts assume this shape.
- **LeetCodeDataset**: problems reshaped into function-plus-docstring form (adapter in `benchmarks.py`); Li's prompts unchanged.
- **LiveCodeBench v6**: 611 stdin/stdout problems wrapped as `solve(stdin: str) -> str`; 444 function-style problems required no wrapping. This is an adaptation of Li's pipeline, not something the paper specifies.

Numbers reported (Claude Opus 4.7 via Vertex; Vertex rejects the temperature parameter for Opus 4.7 so all samples use default settings).

- HumanEval base tests: 92.6% pass@1 / 97.0% pass@10 (1,519 / 1,640 samples).
- HumanEval+ plus tests: 77.4% pass@1 / 84.8% pass@10 (1,270 / 1,640 samples).
- LeetCodeDataset: 63.4% pass@1 / 79.4% pass@10 (1,445 / 2,280 samples).
- LiveCodeBench v6: 55.2% pass@1 / 76.5% pass@10 (5,819 / 10,550 samples).

C ENSURES-BINDING GATE IMPLEMENTATION

The gate is a static AST walk over each generated `Impl.dfy` file. Methods with zero `ensures` clauses fail the gate immediately (matching the $n \geq 1$ conjunct in §3). For every `method` declaration with at least one `ensures` clause, the gate:

1. Parses `Impl.dfy` with Dafny's Python-accessible parser and collects every `ensures` clause as a Dafny expression AST.
2. Locates the paired `Spec.dfy` (same component workspace) and any support files transitively `include`'d from it. Enumerates every top-level `predicate` and `function` declaration in these spec files to build the set $\mathrm{Preds}(S)$ of spec-defined predicate/function symbols.
3. For each `ensures` clause E_i, walks its identifier set and checks whether any identifier resolves to a member of $\mathrm{Preds}(S)$. If any single identifier does, the clause is bound; if none of the `ensures` clauses on the method binds, the certificate is rejected.

Edge cases handled: transitive `include` chains; `function methods` that are both spec and impl symbols (only the spec-side declaration counts for $\mathrm{Preds}(S)$); qualified references (`Spec.PredicateName`); and `forall`/`exists` bindings whose bound variables don't count against the binding requirement.

Runtime: under 10 ms per component on our runs.

D EXISTING SYSTEMS ON THE THREE-SURFACE HIERARCHY

Referenced from §8.

Table 6: What existing pipelines and evaluations report on each surface. Most systems that report Surface A do not report either companion surface.

System	Surface A	Surface B (syntactic binding)	Surface C (semantic fidelity)
Li et al. (2025)	Yes	Not reported	Runtime fixtures only
VeriContest (Xie et al., 2026) end-to-end	Yes	Not reported	Post2Exe on postcondition
AxDafny (Breen et al., 2026)	Yes	Missing/trivial `ensures`, `{:axiom}`, LLM reviewer	Not reported
VCG (our own)	Yes	Ensures-binding gate	Hidden-test proxy on Rust runs; Post2Exe applied on VeriContest LC (7.79% End2End @1, 535/6870)

E PASS TAX BY PRESENCE OF `ENSURES` CLAUSE

Companion to §7.

Table 7: Pass rate by presence of `ensures` clause, Li@Opus, three Python benchmarks. Samples with an `ensures` clause pass hidden tests less often than samples without one on every benchmark. This is not a pipeline defect: a specification that actually constrains the implementation will sometimes reject a "clever" LLM output that happens to work on the tests.

Benchmark (Python)	With `ensures`	Without `ensures`	Delta
HumanEval (base tests)	80.2% ($n = 86$)	94.2% ($n = 1{,}539$)	-14.0 pp
LeetCodeDataset	59.6% ($n = 213$)	66.1% ($n = 1{,}995$)	-6.5 pp
LiveCodeBench v6	53.2% ($n = 1{,}032$)	62.6% ($n = 8{,}416$)	-9.4 pp