SPECIFICATION GROUNDING DRIVES TEST EFFECTIVENESS FOR LLM CODE

Amin Haeri
Model Development Innovation
TD Bank, Toronto, Canada
amin.haeri@td.com

Mahdi Ghelichi
Model Development Innovation
TD Bank, Toronto, Canada
mahdi.ghelichi@td.com

ABSTRACT

Large language models frequently generate code that appears correct on typical inputs yet fails on edge cases, invalid inputs, and specification-defined corner conditions, such as a missing input check or an unhandled boundary case. A popular fix has the model write its own tests and repair until they pass. This helps, but the source of the gain is unclear. Does it come from the tests merely *existing*, or from their grounding in a *specification* of what the code should do? We isolate this factor through a controlled experimental design. Holding fixed everything that normally varies (test count, which model writes the tests, and the repair loop), we change a single line of the test-writer's prompt that controls whether it receives the spec as a checklist of rules. The baseline is no strawman, since it is already told to probe invalid inputs and edge cases. Grounding the tests in the spec produces correct code **+38 percentage points** more often than this strong baseline on each of three Claude tiers (Haiku 4.5, Sonnet 4.6, Opus 4.8), and $+36$ points more often on a held-out set. These results indicate that specification grounding, rather than test quantity, is the primary driver of test effectiveness. Doubling the baseline's budget barely helps, and even combining eight independent ungrounded test suites plateaus far below grounding. An ablation pins the cause to the spec's *content* rather than its format. Given the spec as a plain paragraph, the tester still recovers 27 of 30 bugs; asked to plan tests without the spec, it recovers only 2 of 30. The effect remains robust under stronger baselines. A property-based generator catches 28 of 30 bugs but invents out-of-spec requirements, and a full AlphaCodium-style loop only matches the baseline. It also holds across three model families run full-stack. Beyond Claude, GPT-5.3-codex gains $+28$ points and Gemini 3.5 Flash gains $+19$. A task-level sign test over 18 tasks is significant at $p = 0.002$ (only about a 0.2% chance of a gap this consistent if grounding made no difference). Specification grounding improves both sensitivity and precision: more real bugs are caught *and* far less correct code is wrongly rejected. The false-alarm rate is 0% for the grounded tests versus 33% for the baseline, rising to 68% for the baseline when the oracle is the Python standard library itself. On well-specified algorithmic problems it neither helps nor hurts.

1 Introduction

A modern language model, asked to write a small function, almost always returns something that compiles and handles the common case. Failures frequently arise in boundary conditions and under-specified behaviours. Consider a one-line ticket: *"parse a compact integer range: 'a-b' expands to the inclusive list `[a,...,b]`; a bare 'n' returns `[n]`."* Claude Opus 4.8 writes:

```python
def parse_range(spec: str) -> list[int]:
    spec = spec.strip()
    start, sep, end = spec.partition("-")
    if sep == "":                # bare 'n'
        return [int(spec)]
    a, b = int(start), int(end)
```

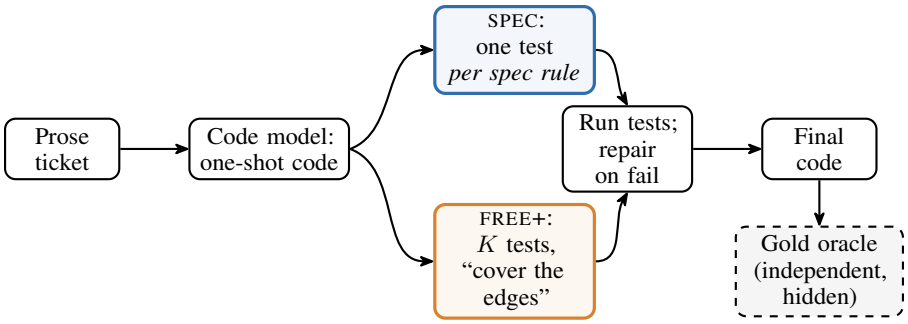

Figure 1: **The experiment in one picture.** A code model writes one implementation from a prose ticket. A *fixed* tester then writes tests under one of two prompts differing in a single line: SPEC gets the spec as a checklist and writes one test per rule (blue); FREE+, our fair baseline, gets only the ticket but is told to "cover the edges" (orange). The tests drive an identical repair loop, and an independent gold oracle the models never see judges the final code. Only the test prompt changes across arms; §4.2 pins the effect on its *content*, not its enumeration.

```
    if a > b:
        raise ValueError(f"start {a} > end {b}")
    return list(range(a, b + 1))
```

This is careful code: it strips whitespace, handles the bare `"n"` case, and even guards the reversed range, so `"5-1"` raises a clear error instead of returning an empty list. Yet corners remain. A malformed `"1-2-3"`, an empty string, and a negative bound such as `"-3-5"` all crash with an opaque `int()` error rather than a clear "malformed range" report; the model itself flagged the negative case as an unhandled assumption. In our experiments *every* model we tried, including the strongest, left omissions of exactly this kind. Such missing input checks are not exotic; they are the everyday gap between "plausible" and "correct," and a leading source of real-world defects.

The field's standard response is to generate tests and let the model iterate: run the tests, feed back the failures, repair. This demonstrably improves code generation, from test-based selection [Chen et al., 2023] to flow-based agents [Ridnik et al., 2024]. But these methods establish only *that* AI-written tests help. They leave open *why*, and the answer determines what a practitioner should actually do.

Two competing explanations. Tests might help for either of two reasons, and the two point to opposite advice. The first is sheer quantity. Running any plausible test sometimes trips a bug, so more tests catch more, and the advice is simply to generate more of them. The second is grounding. Tests drawn from an external specification are aimed at the behaviours that matter, above all the invalid-input and edge cases a model tends to forget, and the advice is to write a specification and turn each clause into a test. These two hypotheses imply different practical recommendations: increasing test volume, or investing effort in explicit specification development. If quantity is the real driver, specifications add little; if grounding is, they are essential.

Our experiment. We isolate the variable. For each task a code model writes one implementation (the realistic "fast pass"; we plant no bugs). A *fixed* tester then writes tests under one of two prompts that differ in a single line. SPEC receives the specification enumerated into discrete rules and writes one test per rule. FREE+ receives only the ticket but is *explicitly told* to test invalid inputs and edge cases. Tester capability, test budget, and repair procedure are held constant across conditions. Correctness is judged not by either set of tests but by an independent, hand-written gold suite that no model ever sees and that is deliberately broader than the rule list. Including FREE+ is what makes the comparison fair, since it already knows it should test edges. Any remaining advantage of SPEC is then about *grounding*, not about reminding a naive tester to think about errors. Figure 1 shows the setup.

We begin with the analysis that most directly identifies the causal mechanism, explaining *why* grounding helps rather than merely that it does. An ablation (§4.2) takes the specification apart. A tester given only a plan that breaks the task into cases, with no spec, catches almost nothing (2 of 30 bugs). Given the spec as a plain paragraph, it catches nearly all of them (27 of 30). Splitting that same spec into a checklist and writing one test per rule adds only a little more (30 of 30). So the spec's *content* is the cause, and the main gain of $+38$ percentage points (pp) follows from it. Every cheaper explanation then fails its own test. Running more ungrounded tests does not close the gap, whether by doubling the budget or by combining several independent test suites (§4, App. I.5). Neither does a generic nudge to

Table 1: The landscape grouped by *where the test author's expected values come from*. Only an external specification grounds those expectations, and none of the prior groups isolates that grounding as the causal lever; SPEC is our arm. The rows elaborate the paragraphs of this section.

Group	Where test expectations come from	Grounding	Isolates grounding?
Ungrounded test-driven (CodeT, AlphaCodium, LEVER)	Model invents each test's expected value	none	no
Self-feedback (Self-Refine, Reflexion, Self-Debugging)	Model critiques its own code	none (self)	no
Weak grounding (property-based, TiCoder)	Author invariants, or a human in the loop	weak	no
External specification (SPEC, ours)	**Enumerated spec rules, one test per rule**	**external**	**yes**

"cover the edges" (the FREE+ baseline), nor a stronger ungrounded generator such as a property-based battery (§4.3) or an AlphaCodium-style flow.

Contributions. We identify specification grounding as the principal mechanism through which AI-generated tests improve code quality, and turn this finding into practical guidance, in five contributions.

- We isolate grounding from test quantity in LLM test-driven repair, holding the tester, budget, and repair loop fixed and changing only whether the tester sees the spec, with correctness judged by an independent gold oracle against a fair, edge-prompted baseline (§3).
- We show that grounding helps in two ways at once, catching more real bugs and raising fewer false alarms, and we trace this to two mechanisms: writing one test per spec rule surfaces the non-obvious invalid conditions a model forgets, and each rule's failure gives a complete repair signal (§4).
- We isolate the contribution of specification *content* from that of task decomposition and coverage planning, through an ablation that separates content from a rule-by-rule split and a chain-of-thought plan (§4.2).
- We bound the claim to specification-completeness defects, not algorithmic logic (§4; App. H).
- We report a capability-substitution result, where grounding a small model matches a larger one run alone, with a token-cost accounting (§4.4; App. E).

2 Background and related work

The work we build on shares one thesis. Test-driven repair helps LLM code generation, and what differs across methods is *where the test author's expected values come from*. We group prior work by the grounding of that signal (ungrounded, self-feedback, and weak grounding), against the external specification we study; none of these prior groups isolates grounding as the causal lever, which is our question (Table 1).

Ungrounded test-driven generation and repair. HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] established functional-correctness evaluation for code models [Rozière et al., 2023]; EvalPlus [Liu et al., 2023] showed thin test suites overstate correctness, motivating our deliberately broad, independent oracle. On the method side, CodeT selects code by dual-execution agreement [Chen et al., 2023], AlphaCodium couples generated tests with an iterative flow [Ridnik et al., 2024], and LEVER trains an execution verifier [Ni et al., 2023]. All treat tests as free-form artifacts whose expected values the model invents, and none ask whether the benefit is the tests' *existence or quantity* or their *grounding*. Determining correct expected outputs remains a manifestation of the "oracle problem" in software testing [Barr et al., 2015]. An ungrounded author must guess, and on non-obvious behaviour it guesses wrong. That is one of the two mechanisms we measure, and grounding supplies a partial oracle for it.

Self-feedback (self-correction and repair). Self-Refine [Madaan et al., 2023], Reflexion [Shinn et al., 2023], and Self-Debugging [Chen et al., 2024] iterate on feedback the model generates about itself; classical program repair predates LLMs [Le Goues et al., 2019]. Two results sharpen our motivation. Huang et al. [2024] show intrinsic self-correction often fails or hurts, and Olausson et al. [2024] pinpoint poor self-feedback as the bottleneck. An enumerated specification can be viewed as a structured form of the stronger, *external* feedback advocated in that prior work, applied at the test-authoring step, and our blind self-refine control confirms it does not substitute.

Weak grounding (property-based testing and TiCoder). Writing behaviour down is old wisdom, from Design by Contract [Meyer, 1992] and Dafny [Leino, 2010] to property-based testing [Claessen and Hughes, 2000]. The last is a spec-to-test pipeline we compare against directly (§4.3) and find only *weakly* grounded: a random-plus-invariant generator catches many bugs but *invents* out-of-spec assertions, the same hallucination an ungrounded edge tester makes. TiCoder formalizes intent through tests with a human in the loop [Lahiri et al., 2022]; we are fully automated and isolate the marginal value of *enumerating* the spec for the tester. We reuse the grounded test-pass fraction as a selective-prediction signal [Geifman and El-Yaniv, 2017], and our independent oracle avoids the self-enhancement bias of LLM judges [Zheng et al., 2023]. Finally, agentic systems on SWE-bench [Jimenez et al., 2024] like SWE-agent [Yang et al., 2024] already iterate against tests; we suggest *where* the tests' expectations come from is a lever orthogonal to better search. A recent *code-as-harness* survey names *oracle adequacy*, that verification is only as good as the oracle behind it, as a key open problem [Ning et al., 2026]; we answer one slice of it.

External specification (our position). Prior work improves LLM coding through better search, more tests, or self-feedback; across all three the load-bearing quantity is the *quality of the feedback signal*. We provide a controlled isolation of its *grounding in an external specification* as the driver, using a fair, edge-prompted baseline and an independent oracle, and show the effect on both sides of the confusion matrix.

3 Method

3.1 Overview

We refer to each test-generation strategy as an experimental *arm*. For each task, a code model writes one implementation. An arm then writes tests for that code, the tests drive a repair loop, and an independent oracle issues the final verdict (Figure 1). Everything is held equal across arms except the test prompt. The two main arms are SPEC and FREE+. SPEC receives the specification split into separate rules and writes one test per rule. FREE+ is a fair baseline that receives only the ticket, but is told to test invalid inputs and edge cases. The SPEC prompt differs from FREE+ in three ways: it adds the content of the spec, splits that content into separate rules, and asks for one test per rule. The ablation in §4.2 separates these three and finds that the added content is what carries the effect. We also run several further controls and stronger baselines, including a doubled test budget, combined test suites, a property-based generator, and an AlphaCodium-style flow, and introduce each one where it is used. The remaining subsections describe the tasks and their hidden oracle, the arms, the models and repair loop, and the metrics.

3.2 Tasks and oracle

Our benchmark has 26 small Python tasks in two groups. The main group is 18 *specification-completeness* tasks, whose plain-language tickets hide real edge cases. It has 8 *core* tasks (semver comparison, money splitting, pagination, median, interval merging, ellipsis truncation, Roman-numeral parsing, and clamping) used as the main testbed (§4.1–4.3), 4 *held-out* tasks used only for replication (§4.4), and 6 *scale-up* tasks added later to enlarge the sample (§4.5). The second group is 8 *logic* tasks, where the hard part is the algorithm rather than input validation; we include them only as a scope control (§4.4). All four subgroups, with their full rule lists, are defined in Appendix A.

Each task comes with four parts: a realistic prose ticket (the only input the code model sees), the specification enumerated into K checkable rules (shown only to SPEC), a trusted reference implementation, and a hand-written *gold* test suite. The gold suite is the oracle that judges final correctness. We validate it against worked examples, never show it to any model, and make it broader than the rule list, so that a passing solution cannot just be "the union of the rules." We write our own tasks instead of reusing a benchmark such as HumanEval+ [Liu et al., 2023] because isolating grounding requires each task to provide *both* an enumerated rule spec *and* an independent oracle; a benchmark's own tests would serve as the spec and confound the comparison. Standard algorithmic benchmarks also rarely contain this bug class, as our HumanEval+ port confirms (App. H). We therefore need under-specified tickets with curated edges to elicit the effect; they are not a setup chosen to favour grounding. A separate set of six tasks (App. I.5) makes the same point in a harder way. Their edge cases come from published standards we did not write (RFC 791, the CSS colour grammar, the ISO ISBN-10 checksum, and similar), so we could not have picked the edges to suit the method. The gap there is even bigger: SPEC catches every bug (100%) while FREE+ catches only 32%, a +68 pp gap. These six tasks are not part of the specification-completeness set or the logic set above. We mean the benchmark as a controlled isolation study, not a leaderboard.

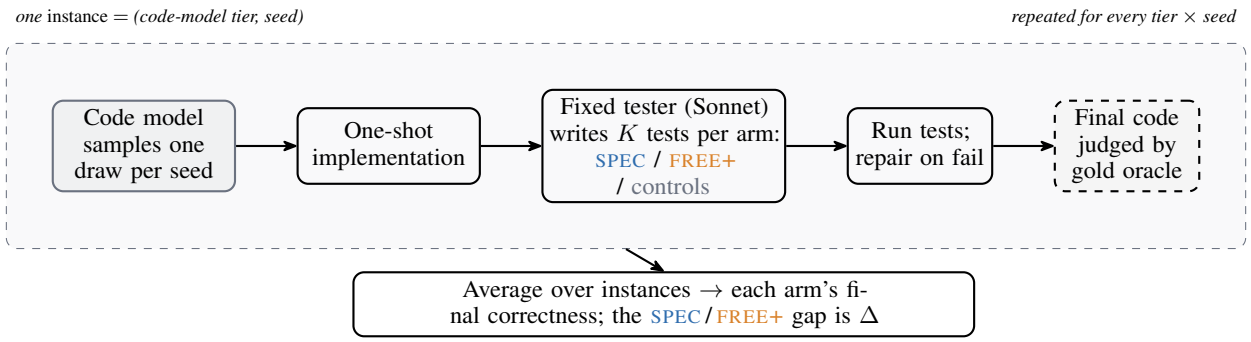

Figure 2: **Draws, seeds, and instances.** The code model is sampled, so each *seed* is one draw and gives a different one-shot implementation (the grey stack on the left). One (code-model tier, seed) pair is an *instance* (the dashed plate). Within an instance every arm tests the *same* implementation and differs only in its test prompt; the inner arm pipeline is the one in Fig. 1. Failing tests drive an identical repair loop, and a hidden gold oracle judges the final code. Here *controls* are the other arms of §3.3: FREE, FREE2K, and the test-free SELF-REFINE and ONESHOT. We repeat the plate for every tier and seed and average over instances, so no single lucky or unlucky draw decides a task; the gap in final correctness between SPEC (blue) and FREE+ (orange) is the main Δ.

3.3 Arms

A code model writes each function once, and every arm works on that *same* one-shot code. Our main comparison is between the two test-writing arms. Both write the same number of tests, K (the number of spec rules for that task), and differ only in their prompt. FREE+, our fair baseline, sees the ticket and is told to test invalid inputs and edge cases. SPEC sees the ticket *and* the spec as K rules, and writes one test per rule. We add four additional control arms:

- FREE: K tests with no edge prompt, the AlphaCodium/CodeT-style baseline [Chen et al., 2023, Ridnik et al., 2024].
- FREE2K: FREE with $2K$ tests, to check whether having *more* tests is what helps.
- SELF-REFINE: the model re-reads and revises its own code, with no tests at all.
- ONESHOT: no checks at all, as a lower bound.

In every arm the tester computes the expected outputs from the ticket or the rules, never from the candidate code, because a tester that sees buggy code tends to treat the bug as intended behaviour. Example prompts are in Appendix B.

3.4 Models, draws, and repair

Three roles use a model: the *code model* writes the function, the *test author* writes the tests, and the *repairer* edits the code from the failing tests. We vary the code model and hold the other two fixed. The code model runs at three tiers of Anthropic's Claude family, small (Haiku 4.5), medium (Sonnet 4.6), and large (Opus 4.8), so we can see the effect across model strengths. The test author and the repairer are always the medium tier (Sonnet 4.6). Fixing the tester this way keeps tester ability constant across arms, so any gap comes from the test *prompt*, not from a stronger or weaker tester. We leave every Claude model at its default reasoning effort, the same in SPEC and FREE+, so it cannot affect the gap. This main study is Claude-only; the cross-vendor check in §4.5 repeats the whole pipeline with OpenAI and Google models in *every* role (code model, test author, and repairer), not just as the code model, and Appendix G lists the generation settings for all of them.

The code model samples its output, so the same ticket can give different code each time we draw from it. We label each draw by a *seed*; a new seed yields a different, independently generated implementation to put under test. We call one (code-model tier, seed) pair an *instance* (Figure 2); the tier is the *code model's*, since the test author and repairer stay at Sonnet throughout. Every arm in an instance is run against that one implementation, so the arms vary in their test prompts (and, for the quantity control, the test budget), never in the code under test. The primary comparison varies the code model over two tiers (Haiku and Sonnet), three seeds each,[1] which gives six instances per arm on each task; §4.4 adds the third code tier (Opus), and the later experiments state their own model and seed counts. Averaging over

[1] A *seed* here is a per-draw index, not an API random-seed parameter; the Anthropic Messages API has none, and we pass no seed to any vendor. We realise the index as a one-line nonce, `(implementation attempt #n)`, appended to the code model's prompt, where `n` is the seed number; the nonce is the only part of that prompt which changes from one seed to the next. Temperature

instances keeps a single lucky or unlucky draw from deciding a task. The repair loop feeds the descriptions of failing tests back to the code model. We give SPEC one repair round, which is enough for it to saturate, and we give the FREE+ baseline up to two. The extra round favours the baseline, so the remaining gap is not explained by SPEC getting more chances to repair. Execution is deterministic inside an isolated subprocess, and final correctness is always the gold oracle's verdict.

3.5 Metrics

The gold oracle judges every arm's final code. We report four quantities, each a simple fraction:

- *Final correctness*: of all instances, the fraction whose repaired code passes the gold suite. This is the bottom-line score, did the arm end with working code?
- *Detection*: of the one-shots that are truly buggy, the fraction the arm's tests catch (at least one test fails). For example, a `clamp` that skips the `lo>hi` check is buggy, and an arm detects it only if one of its tests exercises `lo>hi` and fails on it.
- *False alarms*: of the one-shots that are already correct, the fraction the arm's tests wrongly fail, for example failing a correct `paginate` because the test itself expected the wrong page. A high rate means the tests cannot be trusted.
- *Repair completeness*: of the bugs the arm did detect, the fraction the repair loop then fixed. An arm can catch a bug yet still repair toward the wrong answer when its own expected value is wrong, for example pushing `format_duration` toward `"1h 0m 0s"` when the right answer is `"1h"`.

Our main result is Δ, the gap between SPEC and FREE+ on the same tasks, computed as the SPEC score minus the FREE+ score, so a positive Δ means grounding helped; for instance SPEC at 100% and FREE+ at 60% give $\Delta = +40$ pp (Table 2). We report Δ with a 95% bootstrap confidence interval (a range found by resampling the data many times) and McNemar's test (a significance test for paired yes/no outcomes). The six instances within a task share one code draw and are correlated, so this instance-level test is optimistic; we judge the main claim at the *task* level instead, counting each task once with a sign test (the conservative unit; §4.1). The six instances per task only sharpen each arm's per-task rate, they are not the sample we test for significance. For the abstention analysis (when to trust a piece of code and when to hold it back for a human to check), we use each arm's test-pass fraction as a confidence score and report the area under the risk–coverage curve (AURC). Before we looked at any results, we committed to the decision criteria the study had to clear (the pre-registered go/no-go bars in Appendix D), so the choices could not be tuned to the outcome.

Reproducibility. The study is built from the task specifications, reference implementations, gold suites, the deterministic harness, a provider-pluggable model client (Anthropic/OpenAI/Google), and the raw model outputs. Every reported number is a deterministic function of those artifacts, so the analysis reproduces exactly; inference reproduces up to model sampling. The independent-stack robustness experiments (§I) are built the same way: every appendix number recomputes from their generation and scoring scripts, the frozen rule sets and one-shots, and the raw tester outputs under the same gold oracle.

4 Results

This section reports the results in five parts, then gathers the main numbers into one summary table. §4.1 establishes the primary experimental comparison on the eight core tasks, where our main measure is the paired SPEC−FREE+ gap. §4.2 asks *why* the gap exists and isolates the cause as the specification's content, not its enumerated structure. §4.3 tests the result against stronger automated baselines, a property-based generator and an AlphaCodium-style flow. §4.4 replicates the gap across three model sizes and checks both outcomes, bugs caught and false alarms avoided. §4.5 stress-tests the method on more tasks, two other vendors, and deliberately imperfect specs. §4.6 collects the main numbers in one place.

4.1 The primary experimental comparison

We start with the basic question: on the eight core tasks, does grounding the tests in the spec produce more working code than the fair baseline does? These tasks are the cleanest place to ask it. Each is a small function whose plain-language ticket leaves out a real edge case, so a one-shot draft usually gets the happy path right and the edge wrong; the natural

is a separate, fixed knob: we request 0.8 on every draw. Haiku, Sonnet, and Gemini 3.5 Flash honour it, so their draws differ by both the nonce and sampling; Opus and OpenAI's reasoning-style coder (GPT-5.3-codex) reject sampling parameters, so for them the nonce alone separates the draws.

Table 2: Final correctness and detection by arm on the 8 core tasks. *Final correctness* is the share of drafts that end correct after test-and-repair; *detection* counts the buggy drafts whose tests fail (the arm catches the bug). Two code models write the drafts under test, Haiku 4.5 and Sonnet 4.6; the test author and the repairer are fixed at Sonnet 4.6, so only the code being tested changes across the two models. Each arm writes K tests, where K is the number of specification rules for the task, except FREE2K (which writes $2K$) and ONESHOT (no tests). Runs use one repair round. Detection is out of the 30 buggy drafts among the 48 core instances (8 tasks $\times$ 2 code models $\times$ 3 seeds). FREE is an ungrounded strawman, FREE+ the fair edge-prompted baseline, and SPEC writes one test per rule.

Arm	Final correct	Detection (buggy one-shots)
ONESHOT (no checks)	38%	–
FREE (K)	38%	0/30
FREE2K ($2K$)	42%	3/30
FREE+ (K, edge-prompted)	60%	18/30
SPEC (K)	**100%**	**30/30**

one-shot bug rate here is 62%, so there is plenty to catch. We run every arm on the same one-shot drafts, written by two code models (Haiku and Sonnet) and given one repair round, and report two numbers per arm: the share of drafts that end correct, and how many of the buggy drafts the arm's tests catch. Across the 48 core instances (8 tasks $\times$ 2 code models $\times$ 3 seeds) 30 drafts are buggy, so detection is scored out of 30 (Table 2).

The results reveal a consistent pattern. Doing nothing (ONESHOT) and plain FREE both end at 38%, so tests with no edge focus add nothing. Doubling the test budget (FREE2K) barely helps (42%). The fair FREE+ baseline, told to test invalid inputs and edge cases, recovers part of the gap (60%). SPEC, with one test per rule, reaches 100%. That 100% is a ceiling set by these easy tasks, so from here on we report the paired SPEC$-$FREE+ gap, not the absolute level.

Two alternative explanations are not supported by the data. The first is quantity: maybe SPEC just runs more tests. It does not. SPEC at budget K beats FREE at $2K$ by 58 pp, and even combining eight independent FREE+ suites saturates far below SPEC on the real-world slice (App. I.5). The second is that a generic reminder to test edges should be enough. It is not. SPEC still beats the edge-prompted FREE+ by 40 pp in one round.

The gap is not sampling noise. Against the FREE strawman, McNemar's test finds 30 discordant pairs, all favouring SPEC ($p < 10^{-6}$), with a 95% bootstrap confidence interval on the paired correctness difference of $[+48, +75]$ pp. Against the fair FREE+ baseline, the SPEC$-$FREE+ difference stays above zero on every model (§4.4). The six instances within a task share one draft and move together, so we judge the main result at the task level, counting each task once as the conservative unit. Over all 18 specification-completeness tasks (8 core, 4 held-out, 6 scale-up; §4.5) the mean per-task gain over FREE+ is $+31$ pp. SPEC is strictly better on all nine tasks that show any difference and tied on the rest, a task-level sign test of $p = 0.002$. On the eight core tasks alone the test is only marginal ($p = 0.06$); pooling the held-out and scale-up tasks makes it more reliable and more conservative.

Where the gap lives. The advantage sits on the edges a generic prompt does not think to try. Compare two invalid-input rules. Every arm handles the obvious one: FREE+ catches `paginate`'s "`page<1`" on all six buggy drafts (6/6). It barely catches the non-obvious one, `clamp` with "`lo>hi`" (2/6), which a broad "test the edges" instruction rarely reaches. SPEC catches both on every draft, because one rule names each edge (per-task detail in Table 7, App. E).

4.2 Disentangling grounding from enumeration

SPEC bundles three things: the specification's *content* (grounding), its *decomposition* into discrete units, and a one-test-per-unit *coverage plan*. In principle the gain could come from the structure rather than the content. We separate them with two ablations at fixed budget and tester (Table 3; exact prompts in App. B). PROSE gives the tester the same specification *content* as a single paragraph and asks for K thorough tests (grounding, *no* enumeration). DECOMP gives only the ticket and asks the tester to first decompose the task into K behaviours and write one test each (a chain-of-thought coverage plan, *no* specification content).

The ablation yields a clear separation between grounding and decomposition. DECOMP catches almost nothing (2/30, like FREE). Asking a model to plan its own coverage without the spec reproduces the same happy-path blind spot, so the decomposition/CoT structure is *not* the driver. PROSE catches almost everything (27/30, near SPEC's 30/30). These results indicate that specification content, in *any* form, is the dominant contributor to performance. Enumeration adds only a small margin on top of grounding. The last 3 misses are all on `split_money`: PROSE had the right content

Table 3: Ablation: detection on the 30 buggy one-shots when the tester gets the specification *content* (grounding) and/or is asked to *enumerate* a coverage plan. Grounding drives the result; enumeration adds a small margin.

Condition	spec content?	enumeration?	Detection
FREE	no	no	0/30
DECOMP (CoT plan, no spec)	no	yes	2/30
PROSE (spec as paragraph)	yes	no	27/30
SPEC	yes	yes	**30/30**

Table 4: Detection on the 30 buggy core one-shots across baselines of increasing strength. Grounding leads and, unlike property *invention*, asserts nothing outside the spec (0 vs. 2/15 batteries rejecting the trusted reference).

Tester	Detection	Out-of-spec assertions
FREE (happy-path)	0/30	0
FREE2K (quantity, $2K$)	3/30	0
FREE+ (edge-prompted)	18/30	some (§4.4)
PROPERTY (random + invariants)	28/30	2/15 batteries reject the reference
SPEC (enumerated rules)	**30/30**	**0**

but, writing free-form tests from a paragraph, it did not turn every rule into a test that actually triggers the bug, and it under-covered that one rule. Knowing a rule and writing a test that fires on it are two separate steps, and the second can fail even when the content is present; enumeration closes this small gap by forcing one test per rule. The same rule-to-test gap reappears on the held-out tasks (§4.4). Overall this isolates the mechanism as *grounding* and shows that a "plan-then-test" chain-of-thought tester is no substitute for the specification itself.

4.3 Stronger automated baselines

So far SPEC has only beaten prompt-level baselines (FREE+ and DECOMP). A fair worry is that these are all just prompts, and that SPEC would lose to a real automated test-generation method. So we test it against the two strongest automated methods we know, property-based test generation (first) and a full AlphaCodium-style agentic flow (next), scoring each the same way on the same 30 buggy core one-shots (Table 4). We then look at *why* even these stronger methods fall short: what their tests never check, and why catching a bug is not the same as completing the fix.

Property-based test generation does not beat grounding. The method comes from the QuickCheck/Hypothesis tradition [Claessen and Hughes, 2000]. Rather than write a few hand-picked cases, the tester gets the same ticket (no enumerated rules) and generates a large batch of random inputs, then checks general *properties* that should hold on all of them: for example "the result is always sorted", "encoding then decoding returns the original input", or "an invalid input raises an error". Throwing many random inputs at the function naturally reaches the boundary and invalid-input regions that a hand-picked edge test can miss.

It is genuinely strong: 28/30, far above the edge-prompted FREE+ (18/30). But it does not beat grounding, and the sharper difference is not detection (SPEC is 30/30) but *precision*. Two of the fifteen (task, seed) property batteries (a battery is the full set of property checks generated for one task and seed) wrongly reject the trusted reference implementation, both asserting that `roman_to_int("IIII")` must raise an error. That is a rule the tester *made up*: the spec forbids invalid *characters*, and "IIII" is just a valid way to write four. With no rules to anchor it, the property tester invents expectations, the same failure the edge-prompted tester shows (§4.4); SPEC, held to the stated rules, asserts only what the rules say and invents nothing. So grounding still wins against a strong automated generator, and it avoids the false-alarm risk that unconstrained invention carries.

A full AlphaCodium-style flow does not match grounding. The toughest comparison is a complete agentic flow rather than a single tester. We built an AlphaCodium-style pipeline [Ridnik et al., 2024]: reflect on the requirements, generate tests from that reflection, write a solution, then repeatedly fix code and tests together. We run it on GPT-5.3-codex, a strong coding model, so this code-generation flow gets its best showing. Because this steps outside the Claude-only main study (§3.3), we keep it fair by scoring the flow against SPEC and FREE+ *on that same codex base*, reusing the cross-vendor runs of §4.5. Over the 12 spec-completeness tasks (3 seeds) the flow reaches 72% final

Table 5: Tester accuracy: share of drawn cases whose expected value matches the trusted reference (wrong expectations cause false alarms). Counts are wrong/total. SPEC $\geq$ FREE+ everywhere; the gap widens on trickier specs.

Task set	FREE	FREE+	FREE2K	SPEC
core validation (8)	100.0% (0/228)	98.7% (3/228)	99.8% (1/456)	99.6% (1/228)
held-out (4)	–	100.0% (0/48)	–	100.0% (0/48)
trickier behavioural (3)	–	91.7% (3/36)	–	100.0% (0/37)

correctness, far above the codex one-shot (50%) but exactly the level of the codex FREE+ baseline (72%), and well short of codex SPEC (100%). The reason is the same as before: without the enumerated spec the flow's own tests miss the non-obvious edges, and fixing code and tests together can even rewrite a *correct* test to match buggy code.

What the testers never write (App. E). The mechanism shows up in the tests themselves. FREE writes *zero* invalid-input tests on any task, so the bugs sit exactly where it never looks, while SPEC writes one test per "raise" rule by construction. Grounding systematically converts specification clauses into explicit test coverage.

Catching a bug is not enough: the fix must be complete. A caught bug still has to be fixed, and the fix is only as good as the failing tests that guide it. Even when FREE+ catches a bug, its incomplete set of failing tests gives an incomplete fix: it caught 18 but fully repaired only 11, whereas SPEC caught 30 and repaired all 30 (checked against the gold oracle). So grounding helps at both stages, catching the bug and then repairing it correctly.

Letting the model revise itself with no tests does not help (App. E). We also tried a no-tests arm where the model simply rereads its own code and revises it (the Self-Refine/Reflexion setting). It is unreliable and can even *hurt*: on the small model it broke three correct functions while fixing none (final $25/50/54\%$ vs. grounded 100%). This matches Huang et al. [2024], Olausson et al. [2024]: the missing ingredient is the outside feedback signal the spec supplies.

4.4 Across model sizes: more bugs caught, fewer false alarms

The core result used two models. Does it survive a wider range of coder ability, and does grounding also help on the *other* side, by not failing correct code? We rerun the three arms across three tiers, Haiku, Sonnet, and Opus, giving FREE+ two repair rounds this time to be generous. The gap is stable: SPEC$-$FREE+ $= +38$ pp on every tier. Plain one-shot correctness climbs with model size ($38/38/54\%$ for Haiku/Sonnet/Opus), but FREE+ plateaus at 62% on all three, a ceiling set by the tests rather than the coder, while SPEC reaches 100%. So the smallest model with grounding beats the largest model run one-shot on our benchmark. Detection per tier tells the same story: FREE+ catches 11/15, 7/15, **2/11**, while SPEC catches 41/41.

The other measure: false alarms on correct code. A test strategy can also fail *correct* code, and that is just as damaging: a gate that rejects good code teaches developers to ignore it. On three trickier behavioural specs (Excel column names, title-casing, range collapsing) we checked whether each arm's tests assert *wrong* expected values that would reject a correct implementation. The risk sat entirely on the ungrounded arm. SPEC asserted 0 wrong expected values (0/37) and raised 0% false alarms on correct code; FREE+ asserted wrong expectations on 3/36 cases and falsely rejected correct code on 33% of its checks. The mistakes were systematic, not noise: all three independent FREE+ draws tried the same out-of-domain input, `excel_column(0)`, and each *assumed it should raise*, even though the spec says "positive integer" and the reference returns `""`. Told to "test the edges" without the rules, the free tester steps outside the spec and makes up the expected behaviour; in a real loop it would then "repair" correct code to match that made-up answer. Grounded tests, held to the rules, stay accurate against the oracle. The trade-off is honest: staying inside the rules means SPEC will not probe genuinely important *unspecified* inputs. If you want the test, write the rule.

The cause is visible in *tester accuracy* (Table 5): the share of the tester's cases whose expected value actually matches the trusted reference. SPEC is at least as accurate as FREE+ on every task set, and the gap opens to 100% vs. 91.7% on the trickier behavioural specs, exactly where the ungrounded tester's invented expectations become false alarms. This is not specific to one tester model: sweeping six tester models across two vendors and five capability tiers (App. I.1), SPEC detection stays $\geq$ FREE+ and SPEC false alarms stay $\leq$ FREE+ on every one, with the biggest precision gain on the mid-tier testers.

Grounded tests know when to stop (App. E). If we treat each arm's fraction of passing tests as an "accept this code" confidence score, SPEC is a far better signal for when to trust the code than FREE (risk–coverage AURC 0.257 vs. 0.577). When grounded tests all pass, the code is usually right; when free tests all pass, it is close to a coin flip.

Where it does *not* help. On eight logic tasks, where the trap is algorithmic rather than a missing validation rule, we found *0 natural bugs among 45 valid submissions*. Strong models get well-specified algorithms right in one pass, so there is nothing to catch. We report this null result plainly: the benefit is on specification-completeness defects, not on algorithmic logic. There is a structural reason. To test grounding fairly on a bug class, the tester has to know that class's correct behaviour, which only holds when the behaviour is well-specified, and that is exactly where the coder also tends to get it right.

Held-out tasks. On four fresh held-out specification-completeness tasks (three tiers, one round) the natural bug rate was **100%**: every model dropped a validation rule on every task. The grounding gap held at SPEC$-$FREE+ $= +36$ pp, with SPEC at 61% and FREE+ at 25% (per-model $+25/+33/+50$). This also marks a clear limit: SPEC detected 9/12, not 12/12. On `chunk` the tester turned the rule "`size` ≥ 1" into a test with `size=0`, which already raises, and so missed the `size=`-1 bug that the broader oracle catches. In other words, detection still depends on the tester choosing an input that actually *triggers* the bug, the same rule-to-test gap we saw in §4.2.

4.5 Robustness: more tasks, other vendors, real code, and imperfect specs

The result so far rests on one model family, a fixed task set, our own oracle, and clean specs. Four stress tests ask whether it survives outside those conditions. In every case it does, and where it has a limit that limit is predictable.

More tasks. We added six more specification-completeness tasks (`days_in_month`, `simplify_fraction`, `moving_average`, `format_duration`, `base_convert`, `parse_bool`; App. A) and ran the full pipeline across all three tiers (3 seeds, one repair round; 54 instances). The one-shot bug rate was again high (**78%**). This time *detection* ties (SPEC and FREE+ both 40/42): the bugs are blatant enough that the edge-prompted tester also triggers them, so the whole gap moves to *repair correctness*. Final correctness after test-and-repair is **SPEC 96% vs. FREE+ 78%** ($+18$ pp), almost all of it on `format_duration` (FREE+ 0% vs. SPEC 100%): not knowing the "omit zero-valued units" rule, FREE+ asserted `"1h 0m 0s"` where the answer is `"1h"` and repaired toward the wrong target, while SPEC repaired to the right one. This is the same mechanism as before (grounding $\rightarrow$ correct expectations $\rightarrow$ complete repair), and it lifts the combined task-level test to 18 tasks at $p = 0.002$. Tester accuracy was SPEC 100% (84/84) vs. FREE+ 92.6% (75/81).

Other vendors. To check the effect is not an artifact of one model family, we re-ran the whole pipeline, coder, tester, *and* repairer, on two non-Anthropic models: OpenAI's GPT-5.3-codex and Google's Gemini 3.5 Flash (12 tasks, 3 seeds, 36 instances each). The gap reproduces on both: SPEC$-$FREE+ $= +28$ pp on codex and $+19$ pp on Gemini, against $+38$ on Claude. The whole story holds: FREE catches almost nothing (5/18, 7/24), FREE+ is strong but incomplete (16/18 and 20/24 detection, 72% final correctness on both), and SPEC catches every bug (18/18, 24/24) and repairs the most (100% on codex, 92% on Gemini). Per-vendor settings and per-instance data are in App. G.

Real production code as the oracle. Every result so far uses an oracle we wrote, which invites the worry that we tuned it in grounding's favour. So we re-ran the pipeline on code we did not write: small pure functions reimplemented from under-specified tickets under neutral names, judged against the *real* library as the oracle. These cover the Python standard library (`textwrap.shorten`, `shlex.split`, and six more judged against CPython; App. I.7), utilities vendored verbatim from HuggingFace transformers and NLTK at pinned commits (App. I.8), and the PyPA `packaging` version tools judged against the installed library and its own test data (App. I.9), plus sixteen fresh tasks that include stateful, multi-step programs rather than single functions (App. I.6). The gap reproduces on all of them, in both detection and false alarms. On the transformers/NLTK code SPEC catches 90% of the bugs against FREE+'s 40% ($+50$ pp). On the standard-library and `packaging` code the bugs are easy to trigger but the ungrounded tester *guesses the unstated edges wrong*: it invents expected values that disagree with the real library and rejects correct code 68% and 100% of the time, while grounding restores those expectations and holds false alarms at 0%. A pooled task-level sign test over all 36 independent-stack tasks with a buggy draft favours SPEC ($p \approx 6 \times 10^{-5}$; App. I).

Imperfect specs. The method is only as good as the spec, so we corrupt the spec on purpose on the five buggy core tasks. Dropping the validation rules (an *incomplete* spec) degrades gracefully: detection falls $30/30 \rightarrow 6/30$, but the tests that remain stay 100% accurate, so you lose exactly the coverage of the dropped rules and nothing breaks silently. Adding a wrong rule that *endorses* the bug (a *noisy* spec, e.g. "if $n \leq 0$ return an empty list") keeps detection at 30/30,

Table 6: Results at a glance: final correctness (share of drafts that end correct) for the fair baseline FREE+ and for SPEC across every setting in this section, with the paired gap. SPEC wins in all of them. Two facts sit outside the final-correctness column but point the same way: on the core set SPEC detection is 30/30 vs. FREE+ 18/30 (Table 2), and on correct code SPEC raises 0% false alarms vs. FREE+ 33% (§4.4).

Setting	FREE+	SPEC	SPEC−FREE+
Core, 8 tasks, 2 models (§4.1)	60%	100%	+40 pp
Three model sizes (§4.4)	62%	100%	+38 pp
Held-out, 4 tasks (§4.4)	25%	61%	+36 pp
Scale-up, 6 tasks (§4.5)	78%	96%	+18 pp
Cross-vendor, GPT-5.3-codex (§4.5)	72%	100%	+28 pp
Cross-vendor, Gemini 3.5 Flash (§4.5)	72%	92%	+19 pp

because the correct rule's test still fires, but the cost shows up as precision: tester accuracy drops to 82.8% (15/87 tests disagree with the truth). So the two failure modes stay separate and visible: completeness governs detection, correctness governs precision, and neither fails silently. Two further checks on an independent stack sharpen this: detection tracks the *coverage* of the validation rules rather than their count (App. I.3), and rules a model derives from the ticket by itself recover only part of the benefit, because they restate the stated happy path and miss the unstated edges (App. I.2).

Cost (App. E). Grounding adds almost no compute over the fair baseline. SPEC and FREE+ each make three model calls of similar size (about $6–7 per 1k tasks), so the +38 pp gain comes from aiming the same test budget at the right edges. Getting the same gain from a larger model instead is far more expensive: a medium model with the framework reaches 100%, while a large model run plain reaches only 54%, and closing that gap with model size alone costs about 2.2× the tokens. The last cost is human, and it is where the real effort sits. Writing the rules is light work by count, about 5 rules per task, and a model can draft most of them; but the few validation rules a model tends to leave out (App. I.2) are exactly the ones that catch the bugs. Writing those correctly takes real knowledge of how each edge should behave, which is the hard part (§6).

4.6 Results at a glance

Table 6 gathers the main numbers from every setting above into one place. The pattern is the same in all of them: the fair FREE+ baseline lands part-way, and SPEC ends higher, by between +18 and +40 pp. The gap is smallest where the missing edge is blatant enough that a generic edge prompt also finds it (scale-up) and largest where the edge is easy to overlook (the core and held-out tasks).

5 Discussion

Our main finding is narrow but strong. On tasks where the ticket leaves an edge unstated, turning the spec into one test per rule catches bugs that an equally-informed tester misses, and it does so without failing correct code. Here we explain why a single mechanism produces both effects, why the result is not an artifact of our setup, what could still undermine it, and where the method earns its keep.

A unified mechanism explains both effects. A test can fail in two opposite ways. It can miss a real bug (a false negative), or it can reject correct code (a false positive). Both failures share one root cause: to write a test, the author has to know what the code should do at each edge, and when the ticket leaves an edge unstated an ungrounded tester has to guess. Where it does not think to probe an edge at all, the bug slips through and detection falls. Where it does probe the edge but guesses the wrong intended value, it rejects correct code and false alarms rise. The two errors look opposite, yet they arise from the same missing piece of information: the tester lacks access to the intended behaviour.

The enumerated spec supplies exactly that information. It is an outside signal that states what the code must do, one rule at a time, so the author no longer has to guess. That is why a single signal fixes both errors at once: it points tests at the negative cases the ticket left unstated, which raises detection, and it hands the tester the intended value instead of letting it invent one, which lowers false alarms. Higher detection and fewer false alarms normally trade off against each other; here they move together, because both come from removing the same guesswork. The same logic explains why letting a model revise its own code without tests is weak: as Huang et al. [2024] and Olausson et al. [2024] find, such a model has no outside signal to correct it, and the enumerated spec is exactly the high-quality feedback those methods lack.

The gains are not simple information leakage from the specification. The obvious objection is that SPEC wins only because its per-rule tests restate what the hidden oracle checks. Three things rule that out. First, the comparison that matters is against FREE+, which is also told to test invalid inputs and edge cases, so SPEC's advantage is over an informed baseline, not a naive one. Second, the oracle is independent of the rules and strictly broader, and SPEC is not perfect against it: it scores 9/12 on the held-out tasks (§4.4), whereas if the rules simply were the oracle it would score 100% by construction. Third, "a spec helps" was never in doubt; our contribution is the *size* of that help over an equal-budget, equally-informed tester, and the fact that it improves detection and precision at once.

What could still undermine the result. We keep several worries in view. The bugs are produced by the models themselves rather than planted by us, which keeps them realistic but means we do not control their difficulty; we guard against this by scoring every arm against an independent oracle that is broader than the rule list and checked against an anchor implementation. By "tester" here we mean the model that writes the checks, not that independent oracle. That test author is deliberately strong, and we measure its accuracy (99–100%, with 0 false alarms on the core tasks); a weaker test author might pick inputs that do not trigger a bug, or predict a wrong expected value, so the strength of the test author is a limitation we name and a direction we flag. And the tasks are small pure functions with a claim about specification-completeness defects only; on algorithmic logic we found nothing to catch, and we report that null result openly.

When to use it, and when not. The roughly two extra model calls pay off when a task really has invalid-input behaviour, when shipping a wrong result is costly, and when the spec is a handful of rules. It is a poor fit for tasks with no edges or no agreed specification. For a team shipping LLM-written code, the highest-impact move is cheap: write a short enumerated spec and turn each rule into one check, before reaching for a bigger model or simply more tests. Grounding helps in a second way too: it makes the gate trustworthy. A gate that rejects good code a third of the time trains developers to ignore it, so cutting false alarms matters as much as catching bugs. None of this replaces human review. The method is only as good as the spec it is given, so a high grounded-pass rate gives strong evidence of correctness without proving it (App. E).

6 Limitations and future work

This is a controlled isolation study, so several things are deliberately out of scope.

- **Scale and bug diversity.** The core study is 18 bespoke single functions, with 16 more on an independent stack (App. I.6). These include a stateful, multi-step slice and further slices judged against real production code we did not author: the Python standard library, transformers, NLTK, and PyPA `packaging` (Apps. I.7, I.8, I.9). A larger study would still span hundreds of tasks with algorithmic, semantic, multi-function, and repository-level bugs, not just specification-completeness defects. As an external check, App. H ports the pipeline to HumanEval+ under EvalPlus's own oracle on five model families; the gap vanishes, reproducing our logic-task null on a public benchmark, because well-specified algorithmic problems rarely trigger the defects we target. This motivates a public benchmark of under-specified tickets.
- **Stronger baselines.** We add a property-based generator (§4.3, Claessen and Hughes, 2000), a chain-of-thought DECOMP arm, and a full AlphaCodium-style flow [Ridnik et al., 2024]; coverage- and mutation-guided augmentation remain untested.
- **More model families.** We replicate on GPT-5.3-codex and Gemini 3.5 Flash (§4.5); open-weight models would extend the picture.
- **Richer repair loops.** Search, reranking, reflection, and tool use would go beyond our minimal one-round loop.
- **Repository-level and stateful tasks.** Whole-repository benchmarks (SWE-bench [Jimenez et al., 2024]), agents (SWE-agent [Yang et al., 2024]), and the downstream cost of remediation are all outside our function-level scope.
- **Specification quality at scale.** §4.5 shows graceful degradation: an incomplete spec loses detection in proportion, and a single contradictory rule keeps detection but adds false-alarm risk. Real specs vary far beyond these two perturbations. The binding cost is not writing rules but knowing the correct edge *semantics*: App. I.2 shows a model restating a ticket's happy path but not its unstated error conditions, so deciding whether `excel_column(0)` should raise or return `""` needs domain knowledge, not formatting. Measuring that elicitation cost is a study of its own.

We therefore make a scoped claim: on specification-completeness defects in single pure functions, the effect is large, two-sided, and (per the ablation) driven by grounding. We do *not* claim grounding is the main driver of test-based improvement in every setting. The absolute 100% is an artifact of easy tasks; what matters is the *gaps* and their direction, which stay stable across models, budgets, repair rounds, and baselines.

7 Conclusion

Generating tests helps LLM code generation, but *how* you generate them decides how much. Most one-shot failures on our tasks are specification-completeness bugs: the model handles the stated happy path and drops an unstated edge. What closes that gap is *grounding* the tests in an enumerated specification, one check per rule, rather than writing more tests or adding a generic "test the edges" nudge. Grounding catches more bugs ($+38$ pp over a fair, equally-informed baseline, replicated on Claude, GPT-5.3-codex, and Gemini) and at the same time fails correct code far less often (-33 pp false alarm), because the same signal that aims a test at the right edge also stops the tester inventing a wrong answer. Increasing model scale alone does not yield comparable reliability, and it costs more. The practical implication is straightforward: for code that must be trusted, write down the edge rules and turn each into a check. An important open challenge is obtaining accurate specifications at scale, since the cost is not writing them but knowing the correct edge behaviour in the first place.

Ethics Statement

This work studies automatic test generation for code written by language models. We used small, self-contained programming tasks that we wrote or adapted ourselves. There were no human subjects, no personal data, and no sensitive content. The main risk we see is over-trust: it is easy to mistake a passing test suite for proof that the code is correct. To keep that risk in plain view, we report how often each method wrongly fails correct code (false alarms) next to how often it catches bugs, and we show that writing a *correct* specification still needs a human (§I). Beyond the usual concerns about any code-generation tool, we see no particular misuse potential. To limit the compute we used, we kept sample counts modest and reused saved model outputs instead of re-running them.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024.

Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, 2000.

Yonatan Geifman and Ran El-Yaniv. Selective classification for deep neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Shuvendu K. Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950*, 2022.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

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

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.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

Ansong Ni, Srinivasan Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. LEVER: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning (ICML)*, 2023.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, et al. Code as agent harness: Toward executable, verifiable, and stateful agent systems. *arXiv preprint arXiv:2605.18747*, 2026.

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*, 2024.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with AlphaCodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

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.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, et al. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

A Task specifications

The benchmark has 26 tasks. Most are *specification-completeness* tasks. We list them here by subgroup: core (§A.1), held-out (§A.2), and scale-up (§A.3). The rest are *logic* tasks (§A.4), used only as a scope control. A specification-completeness task is a small function whose plain-language ticket reads as if it covers the behaviour but leaves out real edge cases, such as invalid input, empty input, or boundary values. We test for that gap between the ticket and the full behaviour.

For each task we wrote one specification and split it into a short list of *rules*. A rule is one checkable clause of the specification, for example "empty input raises" or "the output is sorted". We label the rules R1 through RK, where K is the number of rules for that task. SPEC sees this list and writes one test per rule. The list was fixed in advance and did not change between arms or runs. Each entry below gives the function signature, a one-line summary, and its rules. Reference implementations, gold suites, and hand-checked anchors back every task.

A.1 Specification-completeness tasks (core)

- `semver_compare(a,b)`→`int`: compare two SemVer 2.0.0 versions.
  - R1 compare major/minor/patch numerically.
  - R2 a pre-release is lower than the release.
  - R3 pre-release ids compared left-to-right, numeric<alphanumeric.
  - R4 longer prefix wins.
  - R5 build metadata ignored.
- `split_money(total_cents,n)`→`list`: split a money total into n near-equal parts.
  - R1 amounts sum to exactly total.
  - R2 differ by ≤ 1 cent.
  - R3 extra cents go to the first recipients.
  - R4 $n \geq 1$ else raise.
  - R5 total≥ 0 else raise.
- `paginate(items,page,page_size)`→`list`: return one page of a list.
  - R1 pages 1-indexed.

  - R2 page_size≥ 1 else raise.
  - R3 page≥ 1 else raise.
  - R4 past-end returns empty.
  - R5 last page may be short.
- `median(nums)`→`float`: median of a list of numbers.
  - R1 odd count middle.
  - R2 even count average of two middles.
  - R3 sort first.
  - R4 empty raises.
- `merge_intervals(intervals)`→`list`: merge overlapping intervals.
  - R1 output sorted by start.
  - R2 overlapping merged.
  - R3 touching merged.
  - R4 input unsorted.
  - R5 empty returns empty.
- `truncate(text,max_len)`→`str`: truncate text to a maximum length with an ellipsis.
  - R1 fits unchanged.
  - R2 result exactly max_len incl. ellipsis.
  - R3 form is text[:max_len-1]+"…".
  - R4 max_len≥ 1 else raise.
  - R5 max_len$= 1$ returns just the ellipsis.
- `roman_to_int(s)`→`int`: parse a Roman numeral to an integer.
  - R1 symbol values.
  - R2 subtractive pairs.
  - R3 empty raises.
  - R4 invalid char raises.
  - R5 case-insensitive.
- `clamp(x,lo,hi)`→`num`: clamp a number to a range.
  - R1 $x < lo \Rightarrow lo$.
  - R2 $x > hi \Rightarrow hi$.
  - R3 else x.
  - R4 $lo > hi$ raises.

A.2 Held-out specification-completeness tasks

- `parse_range(s)`→`list`: expand a compact integer range.
  - R1 "a-b" inclusive ascending.
  - R2 bare "n" →[n].
  - R3 $a > b$ raises.
  - R4 malformed raises.
  - R5 empty raises.
- `chunk(items,size)`→`list[list]`: split a list into fixed-size chunks.
  - R1 chunks of size.
  - R2 last may be short.
  - R3 size≥ 1 else raise.
  - R4 empty→empty.
- `rgb_to_hex(r,g,b)`→`str`: format an RGB colour as a hex string.
  - R1 "#"+two upper-hex per channel.
  - R2 zero-padded.
  - R3 channels in 0..255 else raise.
- `luhn_valid(number)`→`bool`: check a number with the Luhn checksum.
  - R1 double every second digit from the right, subtract 9 if > 9.

  - R2 valid iff total $\equiv 0 \pmod{10}$.
  - R3 empty raises.
  - R4 non-digit raises.

A.3 Scale-up specification-completeness tasks

- `days_in_month(year,month)`→`int`: number of days in a month.
  - R1 31/30-day months.
  - R2 February 28/29.
  - R3 Gregorian leap rule (div 4, not 100 unless 400).
  - R4 month in 1..12 else raise.
- `simplify_fraction(num,den)`→`tuple`: reduce a fraction to lowest terms.
  - R1 divide by gcd.
  - R2 denominator positive, sign on numerator.
  - R3 zero numerator →(0,1).
  - R4 den$=0$ raises.
- `moving_average(nums,window)`→`list`: sliding-window mean of a list.
  - R1 element i is mean of nums[i:i+window].
  - R2 length len$-$window$+1$.
  - R3 float (true division).
  - R4 window≥ 1 else raise.
  - R5 window$\leq$len else raise.
- `format_duration(seconds)`→`str`: format a duration in seconds as text.
  - R1 split into h/m/s.
  - R2 "<h>h <m>m <s>s" most-significant first.
  - R3 omit zero-valued units.
  - R4 zero→"0s".
  - R5 seconds≥ 0 else raise.
- `base_convert(n,base)`→`str`: convert a non-negative integer to a given base.
  - R1 digits 0-9 then A-F.
  - R2 no leading zeros except n=0 →"0".
  - R3 base in 2..16 else raise.
  - R4 $n \geq 0$ else raise.
- `parse_bool(s)`→`bool`: parse a boolean from text.
  - R1 true/yes/y/1/on→True.
  - R2 false/no/n/0/off→False.
  - R3 case-insensitive.
  - R4 trims whitespace.
  - R5 anything else raises.

A.4 Logic tasks (scope control)

- `is_leap_year`
- `ordinal`
- `mode` (tie→smallest)
- `caesar` (wrap/mod/preserve case)
- `search_insert` (leftmost insertion)
- `excel_column` (bijective base-26)
- `titlecase` (small-word and first/last rules)
- `collapse_ranges`

We list the logic tasks by name only. Their difficulty is in the algorithm rather than in hidden edge cases, so we use them only as a scope control (§4.4). We therefore do not give their full rules here.

B Example prompts

Every arm uses the same fixed tester system prompt, and only the user message changes between arms. The system prompt asks for each test case as structured data: the input arguments, whether the call should raise, and the expected return value. The tester works that expected value out from the specification, not from the code under test. For SPEC the specification is the K rules; for FREE and FREE+ it is the ticket. No arm shows the tester the candidate code, because a tester that reads the buggy code tends to read the bug as intended behaviour. The system prompt is:

```
You are a meticulous test author. Produce test cases as structured data. For each case: args_json
     is a JSON array of the positional arguments; if the call should raise, set expect_error=true
    ; otherwise set expect_error=false and put the CORRECT return value (worked out from the
    specification, not from any code) in expected_json as JSON. Return JSON matching the schema.
```

The user message then changes by arm:

SPEC: *"[ticket]. The specification is these K rules: [R1…RK]. Produce exactly K test cases, one focused case per rule, in order. A rule about invalid input should give a case that expects an error."*

FREE+: *"[ticket]. Produce exactly K test cases that check the function thoroughly. Be sure to include cases for invalid inputs, error conditions, and boundary/edge values, not just typical inputs."*

FREE: as FREE+ without the second sentence.

PROSE (ablation, §4.2): *"[ticket]. Here is the full specification as a single paragraph: [prose spec]. Produce exactly K test cases that check the function thoroughly."* The tester gets the same specification content as SPEC, but as one paragraph and with no one-test-per-rule instruction.

DECOMP (ablation, §4.2): *"[ticket]. First break the task into K distinct behaviours it must satisfy, then write exactly one test case for each behaviour you listed."* The tester plans its own coverage but never sees the specification content.

REPAIR: the code model is shown the current implementation and the failing checks and asked to return the corrected function.

C Representative natural bugs

This section makes the failure mode concrete by showing it in the models' own output. Each listing below is a real one-shot, copied verbatim, next to the gold test case it fails and the rule it breaks. We include them because the "specification-completeness omission" we keep referring to is easier to trust when you can see the actual code: these are ordinary bugs a strong code model writes when a ticket leaves its edge cases unstated, not examples we invented. Every one has the same shape. The happy path is right, and an invalid-input rule is skipped, which is exactly the region a free tester never probes and an enumerated rule names directly.

```python
# split_money  -- fails (10, 0): returns [] instead of raising (R4)
def split(total_cents, n):
    if n <= 0:
        return []
    base = total_cents // n
    remainder = total_cents % n
    return [base + (1 if i < remainder else 0) for i in range(n)]

# paginate  -- fails ([1,2,3], 0, 2): no page / page_size validation (R2,R3)
def paginate(items, page, page_size):
    start = (page - 1) * page_size
    end = start + page_size
    return items[start:end]

# clamp  -- fails (5, 10, 0): no lo>hi check (R4)
def clamp(x, lo, hi):
    return max(lo, min(x, hi))

# parse_range  -- fails "5-1": no a>b check (R3)
def parse_range(s):
    if '-' in s:
```

Table 7: Detection on buggy one-shots, per task (core set; "buggy" out of 6).

Task	buggy	FREE	FREE+	SPEC
semver_compare	0	–	–	–
median	0	–	–	–
merge_intervals	0	–	–	–
split_money	6	0	3	6
paginate	6	0	**6**	6
truncate	6	0	4	6
roman_to_int	6	0	3	6
clamp	6	0	**2**	6
total	30	0	18	30

```
        parts = s.split('-')
        start = int(parts[0]); end = int(parts[1])
        return list(range(start, end + 1))
    else:
        return [int(s)]
```

A further one-shot, `truncate`, shows the same omission, described here rather than listed above. Rule R4 says any `max_len` below 1 must raise, because there is no room for even the ellipsis. The model skips that check and reuses the R3 formula `text[:max_len-1]` plus the ellipsis; at `max_len=0` the slice `text[:-1]` is Python for "all but the last character", so it returns a trimmed string with an ellipsis instead of raising. The strongest model in our study produced omissions of the same kind.

D Pre-registered go/no-go

Before we looked at any results, we wrote down the exact bar the method had to clear to count as a success, so we could not move the goalposts afterwards. The rule was that we would declare **GO** only if every one of these held: the spec-over-free gain $\Delta \geq +10$ pp; the 95% bootstrap confidence interval excludes 0; $\Delta \geq +10$ on ≥ 2 models; a detection advantage $\geq +15$ pp; no more than 5 pp of added false alarms; and SPEC@K beats FREE@$2K$ by $\geq +5$ pp. The observed results clear every threshold.

E Additional results and controls

This appendix collects the supporting checks referenced from the main results, in order: the per-task detection breakdown, how many invalid-input tests each arm writes, a blind self-refine control, a risk–coverage view that uses the test-pass fraction as a confidence score, the cost accounting, a mutation-testing strength check, our AlphaCodium-style flow, and what happens when the tester is shown the candidate code.

Per-task detection. Table 7 breaks core-task detection down by task: FREE catches none, SPEC all, and FREE+ the obvious edges (`paginate` 6/6) but not the non-obvious (`clamp` 2/6).

Negative space: invalid-input tests written per arm. FREE writes *zero* error-expecting tests on every task (so the bugs in the negative space go untested); FREE+ writes some but fewer than SPEC, especially where it under-detects. Counts over six draws: `split_money` 0/6/12, `paginate` 0/8/12, `truncate` 0/4/6, `roman_to_int` 0/6/12, `clamp` 0/2/6 (FREE/FREE+/SPEC).

Blind self-refine (no tests). Each code model re-reads its own one-shot and the ticket and revises it, with no tests (the Self-Refine/Reflexion setting). Final correctness moved from $38/38/54\%$ (plain) to $\mathbf{25}/\mathbf{50}/\mathbf{54}\%$ for the small/medium/large model: on the small model it *regressed three* previously-correct functions while fixing none; on the medium model it fixed three; on the large model it changed nothing net. Blind revision lands far below grounded tests (100%) and is not even monotone, consistent with Huang et al. [2024] (self-correction can degrade) and Olausson et al. [2024] (the missing ingredient is a good external feedback signal, which the spec supplies).

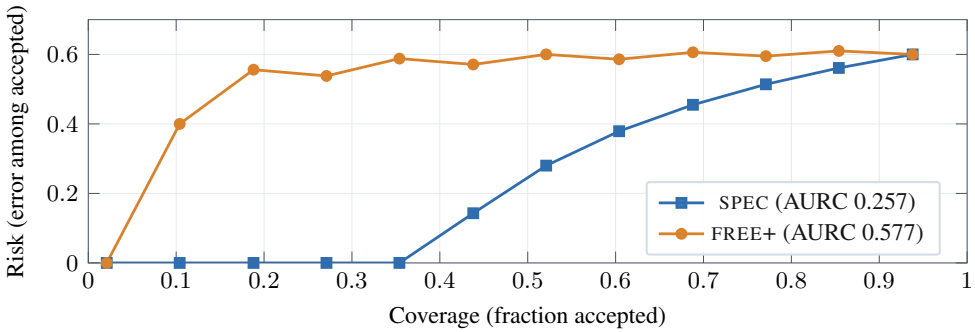

Figure 3: Risk–coverage curves using each arm's test-pass fraction as a confidence score. Lower is better.

Risk–coverage (selective prediction). Figure 3 uses each arm's test-pass fraction as a confidence score for "accept this code." SPEC's curve hugs zero risk out to $\sim 35\%$ coverage; FREE's risk is high immediately (AURC 0.257 vs. 0.577).

Cost. Estimated per-task cost (≈ 4 chars/token; exact call counts; list prices): Sonnet plain 1 call, \$1.84/1k tasks, 38%; Sonnet+FREE+ 3 calls, \$5.83/1k, 62%; Sonnet+framework 3 calls, \$6.84/1k, 100%; Opus plain 1 call, \$3.07/1k, 54%. Grounding costs about the same as the fair baseline yet adds 38 pp; capability substitution costs $\approx 2.2\times$ a plain Opus call.

Mutation testing (suite strength). As a structural strength check, we mutate each trusted reference (relational-op boundary flips, arithmetic-op swaps, integer-constant ± 1, and deletion of `if ...: raise` guards), keep the non-equivalent mutants (those that differ from the reference on the gold suite), and measure what fraction each arm's generated suite kills. Over 10 tasks (6 scale-up + 4 held-out; 147 non-equivalent mutants), mutation score is SPEC 93.9% (138/147) and FREE+ 95.9% (141/147), and it does *not* favour SPEC. The reason is instructive: mutation score rewards a suite for being *sensitive* regardless of whether its expectations are *correct*. On `format_duration` the FREE+ suite kills 18/18 vs. SPEC's 14/18 precisely *because* its out-of-spec assertions ("1h 0m 0s") fail on more mutants, yet those same assertions reject the correct reference and misguide repair (FREE+ final correctness 0% there, §4.5). A suite can be mutation-strong yet wrong; this is exactly why our main metric is end-to-end final correctness, not mutation score.

AlphaCodium-style flow: what we implemented. Our §4.3 flow re-creates the core loop of Ridnik et al. [2024] on single-function tasks: (1) the model reflects on the ticket and lists requirements, including edges and invalid inputs; (2) it generates its own AI tests from that reflection (a $2K$ budget, larger than SPEC); (3) it writes a solution; (4) it iterates up to three rounds, at each step re-running its AI tests and *dual-fixing* both the code and the tests (it may revise a test it judges wrong). It is given only the ticket, never the enumerated rules. This captures the published flow's defining ingredients (problem reflection, self-generated tests, iterative code/test repair) but is *not* the original system: we omit public-test reasoning and multi-candidate solution ranking, and we did not re-tune it against the original's benchmark numbers. So this is a faithful re-creation of the flow's structure rather than a validated port; the full runnable implementation (`alphacodium.py`) is complete and self-contained, so the comparison can be audited or strengthened. The takeaway is robust to these caveats: even with a larger test budget and an iterative dual-fix loop, a flow that lacks the enumerated spec lands at the FREE+ level (72%), not SPEC's 100%.

When the tester sees the code. Our tester writes tests from the ticket/rules, never from the candidate code, because a tester shown buggy code tends to certify the bug as intended. We verified this directly: re-generating tests for 24 buggy one-shots with the candidate code *included* in the prompt, grounded detection is *unchanged* (SPEC 24/24 $\rightarrow$ 24/24) while the ungrounded baseline *collapses to zero* (FREE+ 7/24 $\rightarrow$ 0/24). Shown the code, the free tester writes tests that match the buggy behaviour; the spec's rules still contradict it. This is a third face of the same mechanism: grounding anchors the expected values to the specification, whereas an ungrounded tester anchors them to whatever code it is shown.

Table 8: Per-model results on the 8 core tasks. "det" = detection on buggy one-shots (caught/buggy); "fin" = final correctness.

Model	plain	self-refine	FREE+ det	FREE+ fin	SPEC det	SPEC fin
Claude Haiku 4.5	38%	25%	11/15	62%	15/15	100%
Claude Sonnet 4.6	38%	50%	7/15	62%	15/15	100%
Claude Opus 4.8	54%	54%	2/11	62%	11/11	100%

Table 9: Cross-vendor replication, full-stack (each model is coder = tester = repairer). The grounding gap reproduces on both non-Anthropic families. Claude tiers are Haiku 4.5 / Sonnet 4.6 / Opus 4.8; the non-Anthropic models are GPT-5.3-codex (OpenAI) and Gemini 3.5 Flash (Google).

Family	natural bug rate	FREE+ final	SPEC final	SPEC−FREE+
Claude (3 tiers)	62%/100% (core/held)	62%	100%	+38
GPT-5.3-codex (OpenAI)	50%	72%	100%	+28
Gemini 3.5 Flash (Google)	67%	72%	92%	+19

F Per-model detail

Table 8 gives the full per-model breakdown on the eight core tasks (3 seeds; detection on buggy one-shots; FREE+ with two repair rounds). The grounded arm catches every buggy one-shot on every model; FREE+ detection *falls* as the code model gets stronger (its bugs get subtler), yet SPEC still reaches 100% everywhere.

G Cross-vendor replication

A fair worry is that the effect rides on something specific to Anthropic's models, since the main experiments use Claude for most roles. This section reruns the whole pipeline with non-Anthropic families to rule that out. Each model plays every role *full-stack* (coder, tester, and repairer), so if the grounding gap shows up here it cannot depend on any Anthropic component. Under the hood, the model client routes each request by model id to Anthropic, OpenAI, or Google, and reads back a parsed JSON object and normalised token counts through each provider's own structured-output mechanism, at each model's default settings.

Results (12 specification-completeness tasks, 3 seeds, 36 instances per model). Both non-Anthropic families reproduce the grounding gap (Table 9); detection on buggy one-shots is FREE 5/18 (codex) / 7/24 (Gemini), FREE+ 16/18 / 20/24, SPEC 18/18 / 24/24, the same detection-then-repair structure seen on Claude, where FREE+ detects nearly all bugs but only some of its repairs reach the gold oracle while SPEC detects and repairs (almost) all.

Generation settings and cross-vendor parity. We run code generation as a single quick "fast pass" (the regime that elicits natural specification-completeness bugs), *without* extended reasoning, consistently across families. This matters because Gemini 3.5 Flash enables a "thinking" mode by default that spends far more output-billed reasoning tokens per call than GPT-5.3-codex (≈ 260 output tokens/call, including its light Responses-API reasoning) or the Claude code models (left at their default reasoning effort). Left on, Gemini is the *outlier* in reasoning effort, and because thinking tokens are billed as output it also truncated the JSON response and inflated cost; disabling it brings all three families into the same light-reasoning regime. The reasoning setting is held *constant within each vendor* across SPEC and FREE+, so it cannot bias the grounding *gap* we report; it affects only each family's absolute level, which we do not compare across families. We disclose it for transparency.

H External validity: porting to HumanEval+ (EvalPlus)

To test whether the grounding effect is an artifact of our bespoke suite, we port the *entire* pipeline to a public benchmark judged by a public, independently-authored oracle. Tasks are 24 HumanEval problems sampled evenly across HumanEval/0–163 (a systematic stride, not the easy early ones; restricted to the exact-comparable problems). Both final correctness and detection are judged by *EvalPlus's own* expanded test suite [Liu et al., 2023]: the released canonical solution evaluated over EvalPlus base+plus inputs (≈ 203 cases per task). Neither the benchmark nor the oracle is ours; the only authored artifact is the SPEC rule enumeration, frozen once from each public docstring (mean

Table 10: HumanEval+ port, full-stack, judged by EvalPlus's own oracle. SPEC never outdetects FREE+: the grounding gap is absent because the benchmark elicits logic, not specification-completeness, defects.

Model (full-stack)	one-shot bug rate	SPEC detect	FREE+ detect
GPT-5.3-codex (OpenAI)	8% (6/72)	5/6	6/6
Gemini 3.5 Flash (Google)	3% (2/72)	0/2	0/2
Claude Haiku 4.5 (Anthropic)	12% (9/72)	8/9	8/9
Claude Sonnet 4.6 (Anthropic)	4% (3/72)	0/3	0/3
Claude Opus 4.8 (Anthropic)	3% (2/72)	0/2	0/2
Pooled	**6% (22/360)**	**13/22**	**14/22**

$K = 6.1$). We run full-stack (coder = tester = repairer) on five models: GPT-5.3-codex and Gemini 3.5 Flash via API, and the three Claude tiers via single-pass app inference with no code execution.

The grounding gap vanishes, as our scope predicts. Frontier models rarely produce specification-completeness defects on HumanEval+ (one-shot bug rate 3–12%): the problems are well-specified, with worked examples and valid-input assumptions, so the bugs that do occur are ordinary *logic* errors (e.g. HumanEval/107's `range(1,n)` off-by-one), which any exercising test catches without the spec's enumerated edge rules. Pooled over 360 one-shots, SPEC and FREE+ detect the 22 buggy ones comparably (13 vs. 14, FREE+ marginally ahead), and on no individual model does SPEC outdetect FREE+; across the Claude tiers their false-alarm rates are identical (38/202 each). This is the logic-task null of §4.4 reproduced on a public benchmark with a public oracle. It *bounds* the claim rather than extending it: the specification-completeness bug class the method targets is largely absent from well-specified algorithmic benchmarks, which is exactly why eliciting and measuring the effect calls for realistic, under-specified tickets. The bespoke suite is necessary to exhibit the failure mode, not favorable to the method.

I Additional robustness on an independent inference stack

The experiments in the main paper use direct vendor APIs (Anthropic, OpenAI, Google). As an independent check, we ran nine further experiments on a different inference stack, the VS Code Copilot model API, which serves Anthropic and OpenAI models through one interface. They let us vary the *tester* model widely (§I.1), ask whether the specification must be hand-written (§I.2), measure how much specification is enough (§I.3), reproduce the grounding benefit on new tasks with code from a different stack (§I.4), test tasks whose edges are fixed by an external standard (§I.5), scale the suite up while adding stateful, multi-step tasks (§I.6), reimplement standard-library functions judged against real CPython (§I.7), reimplement functions vendored verbatim from real ML repositories (transformers, NLTK; §I.8), and reimplement the PyPA `packaging` library's version utilities judged against the real installed library (§I.9). The first three reuse the paper's frozen one-shots and judge every test by the same trusted reference and gold oracle, so the numbers are comparable; the last six add fresh tasks and one-shots. One caveat: this API does not enforce structured output, so we add a tolerant parser that also accepts single-quoted JSON. The fixed strong tester (Claude Sonnet 4.6) used in the spec-quality experiments below is unaffected by parsing; only the weakest models in the tester sweep lean on the tolerant parser.

I.1 The grounding benefit holds across tester capability

The main paper varies the *coder* tier (§4.4) but holds the tester fixed at Sonnet 4.6. Here we vary the *tester* instead: we hold the frozen one-shots fixed and re-author tests with six models spanning two vendors and five capability tiers, then score detection on the 30 buggy core one-shots and false alarms on the 18 correct ones (Table 11, Figure 4). Spec detection is at least as high as FREE+ on every one of the six testers (gaps from $+0$ to $+50$ pp), and spec false alarms are no higher than FREE+ on every tester. The precision win is largest in the middle of the range: GPT-5.3-codex and GPT-5.5 drop from 56–67% false alarms under FREE+ to 0% under SPEC. There is an honest floor: the two smallest models (GPT-4o-mini, GPT-5-mini) still emit some wrong expected values even when given the rules, so their spec false alarms stay at 44–67%. Grounding helps once the tester is capable enough to apply a rule correctly, and never hurts.

I.2 What automatically derived specifications miss

A fair objection is that the rules are hand-written, so the method just moves the work. We test whether a model can derive them. For each core task we give Sonnet 4.6 *only the ticket and signature* (never the reference or the gold suite)

Table 11: Tester sweep on frozen one-shots (Copilot API). Detection on 30 buggy one-shots; false alarms on 18 correct one-shots; tester accuracy is the share of drawn cases whose expected value matches the trusted reference. SPEC detection $\geq$ FREE+ and SPEC false alarms $\leq$ FREE+ on every tester.

	detection		false alarm		tester acc	
Tester (weak $\rightarrow$ strong)	FREE+	SPEC	FREE+	SPEC	FREE+	SPEC
GPT-4o-mini	73%	93%	89%	67%	72%	77%
GPT-5-mini	80%	100%	56%	44%	88%	88%
GPT-5.3-codex	77%	100%	67%	0%	86%	99%
GPT-5.5	93%	100%	56%	0%	90%	100%
Claude Sonnet 4.6	100%	100%	11%	11%	97%	98%
Claude Opus 4.8	50%	100%	0%	0%	100%	100%

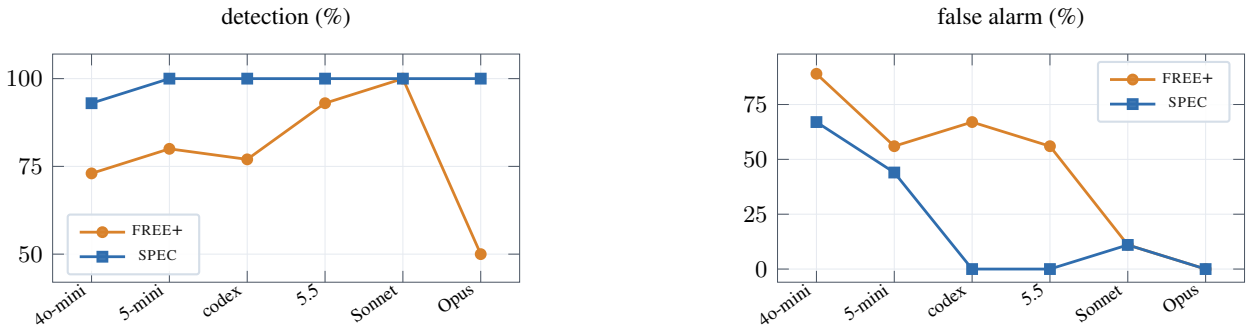

Figure 4: Tester sweep (weak $\rightarrow$ strong), FREE+ vs. SPEC. Left: detection on the 30 buggy one-shots; SPEC (blue) is at or above FREE+ (orange) on every tester. Right: false alarms on the 18 correct one-shots; SPEC is at or below FREE+ everywhere, with the largest precision gains in the middle of the range (codex, 5.5). The two smallest models keep some SPEC false alarms (the capability floor); the largest, Opus, gains the most detection.

and ask it to enumerate the same number of rules as the human spec. We freeze those auto-derived rules, then run tester arms on the frozen one-shots, all with the same fixed tester (Sonnet 4.6) and all judged by the gold oracle: FREE+ (no rules), SPEC@human (the hand-written rules), SPEC@auto (rules a plain prompt derives), and SPEC@auto+ (rules a prompt that explicitly asks for error and boundary conditions derives).

The derived rules *read* well, but they recover only part of the detection (Table 12): SPEC@auto catches 47% of the bugs that SPEC@human catches, and the loss is not spread evenly. Per task, SPEC@auto matches the human spec on `paginate` (18/18) and nearly on `split_money` (15/18), but falls to **0/18** on both `roman_to_int` and `clamp`. Its tests are still accurate (0 false alarms, 96% tester accuracy); they simply do not *trigger* the bug. The reason is precise, and it echoes the paper's central point: the auto-generator paraphrases the behaviour the ticket *states* (the happy path) but does not invent the conditions the ticket leaves *unstated*, which is exactly the specification-completeness bug class. For `clamp`, the human spec's rule "if `lo > hi`, raise `ValueError`" names the bug; the auto spec instead spent its fourth rule on the benign "if `lo == hi`, return `lo`" and never probed `lo > hi`. For `roman_to_int`, all five auto rules describe parsing of valid input, with no rule for the empty string or an invalid character, so the validation bug goes untested. The specification's value is concentrated in the curated edge and error rules the ticket omits; a model can restate a ticket, but restating it does not surface what the ticket forgot.

So we test the obvious fix directly. SPEC@auto+ uses the same generator and the same ticket, but the prompt now asks it to prioritise the error and boundary conditions (§I.3 shows these carry the detection signal). Detection fully recovers: SPEC@auto+ reaches 100%, matching the human spec and catching every bug on `roman_to_int` and `clamp` that the plain version missed. But the recovery comes at a price. Forced to enumerate edges, the generator now *invents* error semantics the ticket never pins down, and some are wrong: its `roman_to_int` rules declare the parser case-sensitive, so a test rejects `roman_to_int("iv")`, which the reference accepts. False alarms jump from 0% to 33% and tester accuracy falls to 93%. So the detection half of the specification automates with the right prompt, but the precision half does not: getting the edges *right*, not merely listing them, is the curation a human spec still supplies. Pushed to cover the edges blind, the auto-generator re-enters the same invent-the-oracle failure mode as the ungrounded tester (§4.4).

Table 12: Auto-derived vs. hand-written specifications, same fixed tester (Sonnet 4.6) on the frozen one-shots, judged by the gold oracle. A plain prompt (SPEC@auto) stays accurate but recovers only 47% of SPEC@human detection, missing the *unstated* error conditions on `roman_to_int` and `clamp`. Prompting for the edges (SPEC@auto+) recovers all detection but invents some wrong error semantics, so false alarms rise to 33%: detection automates, precision still needs human curation.

Tester arm	detection	false alarm	tester acc
FREE+ (no rules)	100%	0%	98%
SPEC@human (hand-written)	100%	0%	99%
SPEC@auto (plain prompt)	47%	0%	96%
SPEC@auto+ (edge-prioritised)	100%	33%	93%

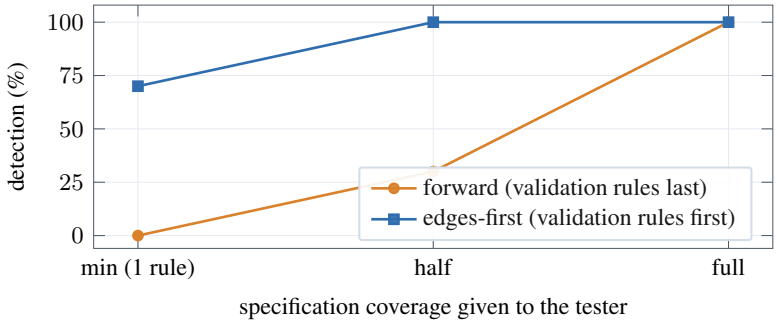

Figure 5: Specification dose-response: detection on the 30 buggy core one-shots as the spec tester is given a growing subset of rules. Forward order rises $0/60 \to 18/60 \to 60/60$ as coverage grows; edges-first already reaches $42/60$ (70%) with a single validation rule and $60/60$ by half coverage. The *content* of the validation rules, not the rule count, drives detection.

(With this strong tester FREE+ detection also ceilings at 100%, so the informative contrast is among the spec variants, not the FREE+ baseline.)

Spec authoring cost. The specs are short: 4.75 rules per core task on average (38 rules over 8 tasks), and an LLM drafts most of the text for free (the SPEC@auto column). What an auto-draft cannot supply is the small set of *error and boundary* rules it omits or invents: on the eight core tasks a human need add or correct just five such rules (one on `clamp`, two on `roman_to_int`, one on `truncate`, one on `split_money`), and those are exactly the four tasks where SPEC@auto detection collapses (Table 12). So the human effort is small in volume but high in leverage: curate a handful of failure-mode rules, not write the whole spec. Those rules come from sources a developer already has: the invalid-input policy, type and domain preconditions, boundary conventions, and any external contract (an RFC, a format spec, a library's documented errors), which is exactly where the real-world slice (§I.5) takes its edges.

I.3 Detection rises with specification coverage

Finally we measure dose-response: how detection scales with how much of the specification the tester is given. For each core task we run the spec tester with a growing subset of its rules, under two orderings: *forward* (rules in their natural order, where validation rules usually come last) and *edges-first* (validation and error rules moved to the front). The pattern is clean (Figure 5). Under forward order, detection rises monotonically with coverage, $0\% \to 30\% \to 100\%$: one happy-path rule catches nothing, and you need the full specification to catch every bug. Under edges-first order, detection is already 70% with a *single* rule and 100% at half coverage. So detection scales with coverage of the *right* rules, not with the raw rule count: the validation rules that name the failure mode carry the signal, which is why a complete specification, or at least one that front-loads its error conditions, drives detection.

I.4 Replication on new tasks with a fresh code stack

The experiments above reuse the paper's frozen one-shots, so a fair worry is that the effect is tied to our task suite or to how that original code was written. We therefore authored five *new* specification-completeness functions after the fact (`to_ordinal`, `time_to_minutes`, `nth_triangular`, `parse_hex_byte`, `roman_from_int`), each with an anchor-

Table 13: Five new validation tasks, fresh one-shots from three Copilot coders (30 one-shots, 67% buggy), fixed Sonnet 4.6 tester, judged by anchor-validated gold oracles. SPEC detects +38 pp more of the buggy one-shots than FREE+, with no false alarms on either arm.

Tester arm	detection	false alarm	tester acc
FREE+	57% (34/60)	0% (0/30)	98%
SPEC	95% (57/60)	0% (0/30)	100%

Table 14: Real-world slice: six tasks whose edge behaviour is fixed by external standards (RFC, CSS, ISO ISBN-10, common library conventions); fresh one-shots from three Copilot coders (34 one-shots, 85% buggy); fixed Sonnet 4.6 tester, judged by anchor-validated oracles. The +68 pp detection gap is the largest in the paper.

Tester arm	detection	false alarm	tester acc
FREE+	32% (28/87)	0% (0/15)	91%
SPEC	100% (87/87)	0% (0/15)	100%

validated reference oracle, and generated fresh one-shots for them with three Copilot coder models (GPT-5-mini, GPT-5.3-codex, Sonnet 4.6; two seeds each, 30 one-shots). The new tickets elicit the same defect class: the one-shot bug rate is 67% (20/30), with coders omitting the unstated error and boundary behaviour (for instance, every `nth_triangular` one-shot forgot to reject a negative index). On the buggy one-shots a fixed Sonnet 4.6 tester detects 95% (57/60) under SPEC versus 57% (34/60) under FREE+, a +38 pp gap, with zero false alarms on either arm (Table 13). The effect reproduces at full strength on tasks we wrote after the study and on code from a different inference stack, not only on the original frozen suite.

I.5 A real-world slice: tasks fixed by external contracts

A sharper test of external validity uses tasks whose correct edge behaviour is set by a *published standard or a widely used library*, not by us, so the oracle cannot be "favourable by design." We authored six such tasks (strict dotted-quad IPv4 parsing per RFC 791, CSS hex-colour parsing, ISO ISBN-10 checksum validation, CamelCase-to-snake conversion, compact duration parsing, and URL slugification), each with an anchor-validated reference that matches the external contract and an under-specified ticket that states the happy path but omits the edges. Fresh one-shots from three Copilot coders (two seeds; 34 valid one-shots) reproduce the defect class at full force: the one-shot bug rate is 85% (29/34), with coders omitting the documented edges (every `parse_ipv4`, `validate_isbn10`, and `parse_duration` one-shot is buggy). A fixed Sonnet 4.6 tester then detects **100**% (87/87) of the bugs under SPEC versus **32**% (28/87) under FREE+, a +68 pp gap (the largest in the paper), with zero false alarms on either arm and tester accuracy rising 91% to 100% (Table 14). The gap is *larger* here than on the core suite precisely because real contracts hide more unstated edges (leading-zero IPv4 octets, an acronym run in CamelCase such as `HTTPServer`, a bare number with no duration unit) that an edge-prompted but ungrounded tester does not think to probe. The effect is not an artifact of tasks we designed to show it.

Best-of-N: more sampling does not substitute for grounding. A natural rescue for the ungrounded arm is to draw *many* FREE+ suites and union them. Drawing eight independent FREE+ suites per task and unioning the first k, detection on the 29 buggy one-shots saturates almost at once: $9/29$ at $k{=}1$, then $15/29$ (52%) from $k{=}2$ onward, *flat* all the way to $k{=}8$ (Table 15). The 14 bugs FREE+ never catches are the most contract-specific edges (leading-zero IPv4 octets, acronym runs in CamelCase-to-snake conversion, a bare number with no duration unit): an ungrounded tester re-probes the same obvious cases and never invents the missing rule, so extra samples add nothing, while SPEC reaches 100% with a single grounded draw.

Table 15: Best-of-N on the real-world slice (29 buggy one-shots): detection when the first k independent FREE+ suites are unioned (caught out of 29). Ungrounded sampling plateaus at 52% from the second draw; a single grounded SPEC draw reaches 100%.

	FREE+ $k{=}1$	FREE+ $k{=}2$	FREE+ $k{=}4$	FREE+ $k{=}8$	SPEC
Detection	31% (9)	52% (15)	52% (15)	52% (15)	**100% (29)**

Table 16: Scaling and diversifying the suite (independent Copilot stack; two Copilot coders; fixed Sonnet 4.6 tester, 2 draws; gold-oracle judged). DIV is six stateful or multi-step tasks (the entry point consumes an operation sequence); N is ten more single-function tasks. Detection is on buggy one-shots, false alarms on correct ones.

Slice	arm	detection	false alarm	tester acc
DIV: stateful/multi-step	FREE+	83% (38/46)	0% (0/2)	97%
(23 buggy one-shots)	SPEC	100% (46/46)	0% (0/2)	87%
N: more single-function	FREE+	96% (46/48)	6% (2/32)	94%
(24 buggy one-shots)	SPEC	100% (48/48)	0% (0/32)	100%

I.6 Scaling up and diversifying the task suite

Two external-validity questions remain: whether the effect is tied to the *size* of the suite, and whether it depends on its tasks being *single pure functions*. We add sixteen fresh tasks, authored after the study and anchor-validated, and run the full pipeline on the independent Copilot stack (two Copilot coders for the one-shots; fixed Sonnet 4.6 tester), growing the treatment base from 18 to 34. Ten are more single-function specification-completeness utilities (`iso_weekday`, `parse_int_base`, `validate_isbn13`, `snake_to_camel`, and so on); six are *stateful or multi-step*: the entry point consumes an *operation sequence* and correctness depends on accumulated state and cross-operation contracts (a bank ledger that must reject overdrafts, a bounded stack that must reject overflow and underflow, a token-bucket limiter, an id pool, a phone normaliser, an HTTP query parser), a different bug surface from single-call validation. The natural one-shot bug rate is high on both (60% single-function, 96% stateful, 73% overall), so the defect class transfers to the new tasks.

The grounding effect reproduces on both slices (Table 16). On the stateful/multi-step slice, detection rises from FREE+ 83% to SPEC 100% (+17 pp): grounding closes the cross-operation contracts (overdraft, overflow, exhaustion) that an edge-prompted but ungrounded tester only probes in part. On the ten new single-function tasks detection is near the ceiling for both (FREE+ 96%, SPEC 100%), and the benefit lands where the scale-up section predicted (§4.5), in *precision*: FREE+ falsely rejects correct code on 6% of checks at 94% tester accuracy, while SPEC holds 0% and 100%. One honest wrinkle: on the stateful tasks the grounded tester's own accuracy dips to 87%, because the exact output *sequence* for a multi-operation input is hard to work out by hand; this does not turn into a false alarm (the single correct stateful one-shot is never wrongly rejected) and detection still reaches 100%, because the error-contract cases that carry detection (§I.3) are the easy ones to get right. So the effect holds on a larger suite and on stateful, multi-step tasks, not only single pure functions.

I.7 A real external oracle: reimplementing the standard library

The sharpest answer to "the oracle is favourable by design" is an oracle we did not write. We take eight Python standard-library utilities (whitespace-collapsing truncation, word titlecasing, shell-style argument splitting, sign-aware zero-padding, URL percent-encoding, the last path segment, tab expansion, and path normalisation) and ask a coder to reimplement each from an *under-specified* ticket under a *neutral* name, so it cannot simply recall the library. Correctness, detection, and false alarms are all judged against the *actual CPython function* (`textwrap.shorten`, `shlex.split`, `str.zfill`, and so on): the oracle is battle-tested production code, and the SPEC rules restate the library's own documented behaviour, not a contract of ours. The bug class survives this real oracle, with a 47% one-shot bug rate (a first version that *named* the library function was reproduced almost perfectly, the EvalPlus null again, which is why the tickets are under-specified).

Three results follow (Table 17). *Grounding* (#1): against the real oracle, SPEC detects 100% of the bugs at 0% false alarms, while the edge-prompted FREE+ detects 83% but falsely rejects correct code 68% of the time. Not knowing the exact documented behaviour (how `textwrap.shorten` collapses whitespace, how `shlex` treats a backslash), the ungrounded tester invents wrong expected values and rejects implementations that in fact match CPython; this is the largest false-alarm gap in the paper outside the saturated `packaging` slice (App. I.9), on an oracle we did not author. *Coverage is not grounding* (#3): a third arm prompted to *maximise branch and path coverage* catches only 37%, *below* even FREE+ – it writes accurate but shallow cases (94% accuracy, 0% false alarms) that execute paths without asserting the contract value on the error conditions, so it misses the specification bugs. *A weak tester* (#2): repeating with a cheap GPT-4o-mini tester, the detection gap *widens* to SPEC 93% vs FREE+ 53% (+40 pp), and grounding still cuts false alarms (47% → 35%), though a weak tester keeps a precision floor (it mis-applies even a correct rule), consistent with the tester sweep of §I.1.

Table 17: Standard-library-oracle slice: eight utilities reimplemented from under-specified tickets under neutral names (15 buggy, 17 correct one-shots), judged against the real CPython function. The ungrounded arms invent wrong expectations for genuine library behaviour (high false alarms); a coverage-maximising prompt covers paths but misses the contract.

Tester	arm	detection	false alarm	tester acc
Claude Sonnet 4.6	FREE+	83% (25/30)	68% (23/34)	80%
Claude Sonnet 4.6	SPEC	100% (30/30)	0% (0/34)	94%
Claude Sonnet 4.6	coverage-maximising	37% (11/30)	0% (0/34)	94%
GPT-4o-mini	FREE+	53% (16/30)	47% (16/34)	83%
GPT-4o-mini	SPEC	93% (28/30)	35% (12/34)	86%

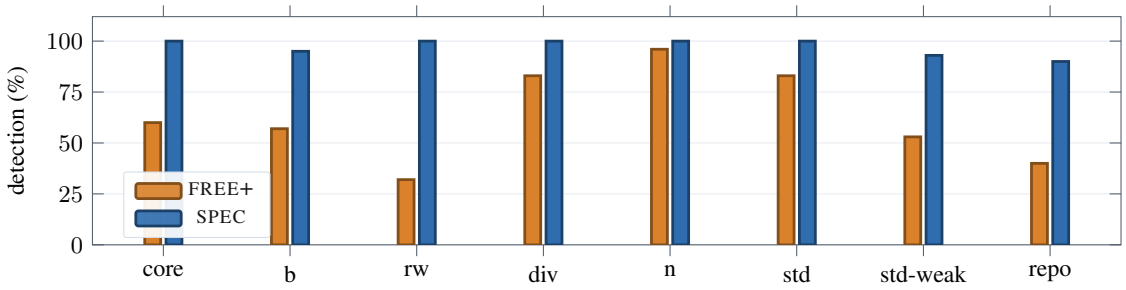

Figure 6: Detection (buggy one-shots caught) by arm across the study's slices: the main core set and the independent-stack slices (`b`, `rw`, the `div` and `n` slices of App. I.6, the `std` slice and its weak-tester variant, and `repo`, the real transformers/NLTK oracle of App. I.8). SPEC (blue) reaches or nears 100% on every slice; FREE+ (orange) varies with how obvious each slice's edges are, but is never higher.

A pooled task-level test. Across all 36 independent-stack specification-completeness tasks that produced a buggy one-shot (§I.4, §I.5, §I.6, this slice, App. I.8, and App. I.9), SPEC detection is $\geq$ FREE+ on *every* task, strictly greater on 14 and tied on 22; a one-sided sign test gives $p \approx 6 \times 10^{-5}$. This complements the main study's task-level test (final correctness, 18 tasks, $p = 0.002$) with a detection-based test on an independent stack, and Figure 6 shows the detection gap slice by slice.

I.8 Real third-party repositories as the oracle: transformers and NLTK

The standard-library slice uses CPython as the oracle; we push the same idea to real, widely-used *machine-learning* repositories. We take seven small pure-Python utilities from HuggingFace `transformers` (the BERT tokenizer's `whitespace_tokenize`, `_run_strip_accents`, `_run_split_on_punc`, `_clean_text`, and the `_is_punctuation` classifier) and NLTK (`edit_distance`; `reduce_lengthening`), vendor each *verbatim* at a pinned commit (transformers `c21da1b5`, NLTK `16a32d68`), and use that real function as the oracle. A coder reimplements each from an under-specified, neutrally-named ticket; the one-shot bug rate is 36%, concentrated on the genuinely edge-laden tokenizer functions (`_clean_text`'s control-versus-whitespace rule, `_is_punctuation`'s ASCII-range definition, punctuation splitting), while textbook utilities like `edit_distance` are reproduced correctly.

Against this real-repository oracle the grounding gap is large (Table 18): SPEC detects 90% of the bugs versus FREE+'s 40%, a +50 pp gap, with no false alarms on either arm. The ungrounded tester writes accurate but shallow cases that exercise the happy path without probing the library's documented edge contracts, so it misses the deviations that matter; the enumerated rules, restating the repository's own behaviour, aim tests straight at them. These tasks are included in the pooled task-level test above and in Figure 6.

I.9 A real package-index library as the oracle: PyPA `packaging`

The transformers/NLTK slice shows ungrounded testing *missing* bugs. A real library can also draw out the opposite failure: an ungrounded tester that guesses the under-specified edges *wrong* and so builds an oracle that rejects correct and buggy code alike. We take four version-handling utilities from the PyPA `packaging` library (name canonicalisation, version canonicalisation, version comparison, and version validity) and use the *real installed library* (version 26.2)

Table 18: Real third-party ML-repository oracle: seven pure-Python utilities vendored verbatim from HuggingFace transformers and NLTK at pinned commits (28 one-shots, 10 buggy, 18 correct), judged against the real functions. SPEC detects $+50$ pp more of the bugs than FREE+.

arm	detection	false alarm	tester acc
FREE+	40% (8/20)	0% (0/36)	100%
SPEC	90% (18/20)	0% (0/36)	95%

Table 19: Real package-index-library oracle: four version utilities reimplemented against the installed PyPA `packaging` library (version 26.2; gold data from the project's own tests at commit `e80f70f8`). 16 one-shots (14 buggy, 2 correct). Detection is at ceiling for both arms; the gap is in precision: FREE+ writes wrong expected values and rejects every candidate, while SPEC accepts the correct ones and rejects the buggy ones.

arm	detection	false alarm	tester acc
FREE+	100% (28/28)	100% (4/4)	85%
SPEC	100% (28/28)	0% (0/4)	100%

as the oracle, with the gold inputs and expected values copied verbatim from `packaging`'s own test suite at a pinned commit (`e80f70f8`). A coder reimplements each from an under-specified, neutrally-named ticket. The bug rate is high ($14/16$): the tickets name the common case but leave PEP 440's edges (non-zero epochs, local versions, pre/post/dev tags, a leading `v`, and separator and newline handling) implicit, so the coders write naive dotted-integer logic that misses them.

On this slice detection saturates (both arms catch all 14 buggy one-shots), so the arms separate on *precision* instead (Table 19). Asked to test edges the ticket never pins down, FREE+ writes expected values that disagree with the real library 15% of the time; for instance it assumes an empty or malformed version string raises, when `packaging` returns it unchanged. A suite with wrong expected values fails every candidate it judges, so FREE+ flags both correct one-shots as buggy (a 100% false alarm rate) and cannot tell correct code from buggy at all. Restating the library's own rules fixes every expected value (100% tester accuracy), so SPEC keeps the full detection while dropping false alarms to 0%: it accepts both correct candidates and rejects all fourteen buggy ones. On a real third-party oracle, grounding turns a suite that rejects everything into one that classifies perfectly. These four tasks are included in the pooled task-level test above.