Combined but Not Reached: AI-Generated Tests for Requirement Interactions

Shirshak Jung Thapa
Florida Atlantic University
Boca Raton, Florida, United States
sthapa2026@fau.edu

November 2026. DOI: 10.5281/zenodo.24482952. License: CC BY 4.0.

***Abstract*—Some defects appear only when two requirements hold at once. We asked a coding agent to write tests for eight such historical defects from five open-source projects, from neutral requirements and the code before the repair. Each defect was attempted three times with a plain prompt and three times with an added instruction to exercise the requirements together. For every generated test we recorded three things: whether it combined the two requirements, whether it reached the code path that triggers the defect, and whether it detected the defect. Of 102 valid tests, 40 combined the requirements, 7 reached the triggering path and 6 detected the defect. Reach was the bottleneck. Of the 29 scored suites that missed their defect, 21 combined the requirements without reaching the path, and this happened in every defect that went undetected. In one Celery test an in-memory backend bypassed the faulty message-broker path, so a correct assertion passed on both revisions. The added instruction raised the number of tests that combined the requirements from 16 to 24 and left detection unchanged at 3 of 15 runs per prompt. A test generator should check that its fixture can reach the implementation path behind the behavior it asserts.**

***Index Terms*—software testing, test generation, requirement interactions, large language models, coding agents**

I. Introduction

Consider a task queue that can run a task eagerly, inside the caller, and that lets a task ask to be retried. One test can check eager execution. A second can check that a worker signals a retry. Both can pass while a retry requested during an eager call returns control to code that should have been interrupted. The defect lives in the combination. A useful regression test has to build that combination and assert what the caller sees.

AI coding agents now write many tests, so it matters whether they build such combinations and what happens when they do. We studied eight historical defects of this kind. For each, two documented requirements pass alone on the code before the repair, and a test that combines them fails. A coding agent received the requirements and the pre-repair code, and wrote tests. It did not receive the bug report, the repair or our own test.

We asked three questions about every generated test. Did it exercise the two requirements together? Did execution reach the state that triggers the defect? Did a valid assertion tell the faulty behavior from the repaired one? These are the steps of the fault-detection chain that testing research has long described: a test must execute the fault, the fault must corrupt the state, and the corruption must reach an output that the test checks [1]. We apply that chain to generated tests and add one step in front of it, the choice of what to combine.

The answer is short. The agent often chose the right combination and rarely reached the faulty path. Of 102 valid tests, 40 combined the requirements, 7 reached the triggering path and 6 detected the defect. Once a test reached the path it almost always detected the defect. An instruction to combine requirements produced more combining tests and no more detections.

II. Related work

Combinatorial testing separates the choice of combinations from the statement of their expected outcomes [2]. The oracle problem concerns the second part [3]. Our tests mostly failed earlier, before the oracle mattered.

Defects4J supplies real repairs against which generated tests can be judged [4]. TestPilot gave a model implementation context and execution feedback to generate JavaScript unit tests [5]. LIBRO generates tests from bug reports [6], and SWT-Bench asks whether a generated test reproduces a reported issue [7]. In both, the reported symptom says what to test. Here the agent saw neutral requirements only, so reproduction rates from those studies are not comparable with ours.

Test doubles explain part of what we observed. Developers mock dependencies that make testing hard, and report that keeping a mock's behavior in step with the real class is difficult [8]. A generated test that swaps a message broker for an in-memory substitute makes the same trade without saying so.

III. Study design

Defects. We searched pull requests created and merged in 2015–2024 in eight GitHub repositories for eight terms (cache, permission, filter, transaction, retry, serializer, session and signal). Sixty-four searches gave 3,960 distinct references. A rule fixed in advance took the first 240 in an order interleaved across repositories. Reading the issue and pull-request texts gave 55 plausible references. The first 24 plausible representatives of distinct fault families formed the qualification queue. Review stopped at slot 16 with eight accepted defects, four blocked, two infeasible and two rejected. The eight span five repositories and four mechanism classes set in advance.

A defect was admitted when it had two separately documented obligations and an assertion about an observable consequence that follows from the requirements. We ran each obligation alone and both together, on the code before and after the repair. Both isolated tests had to pass on each revision. The combined test had to fail at its assertion before the repair and pass after it. Repetition from a fresh state gave 12 observations per defect, 96 in all.

Generation. Prompt A gave neutral requirements, the pre-repair source and a common submission contract. Prompt B added one paragraph:

> In addition to checking each described requirement separately, deliberately consider cases in which the separately described capabilities are enabled or exercised together. Derive every expected outcome from the supplied requirements and library semantics; do not infer any hidden defect or invent behavior.

Each defect had three matched repetitions of A and B, 48 attempts in all. A hash fixed in advance set the order within each pair. Each attempt used a fresh GPT-5.6 Sol session at medium reasoning effort, its own workspace and a 1,800-second limit. A broker restricted tools to approved file operations and syntax checks, so the agent could not run its tests. We froze the first complete submission. For the 42 complete submissions, a replay matched all 903 broker calls and verified file hashes, deadlines and shutdown.

Scoring. The unchanged suites were run against pinned revisions. A detection required a valid oracle, a relevant assertion failure before the repair, a pass after it, and evidence that the interaction caused the difference. An uncaught exception alone did not count. Two graders worked from records keyed by opaque identifiers. Their verdict categories agreed on all 131 tests. A third reviewer resolved differences in the supporting fields before prompt labels were joined to scores.

IV. Results

A. Detection

Table I gives every scheduled run. Thirteen scores are missing. Six Django attempts produced no complete submission, and seven Rails suites could not run because a library was absent from the evaluation environment. The remaining 35 runs were scored.

The two prompts detected the same defect. In the five defects with all three pairs scored, each prompt detected the target in 3 of 15 runs. All six detections are the eager-retry defect, which both prompts found in every repetition. Over all scored runs the counts are 3 of 18 for A and 3 of 17 for B.

The analysis plan, fixed before generation, defined the effect as the mean over eight defects of the difference in detections per defect, and required all three pairs of a defect to be scored. Five defects met the rule, and each difference is zero. The eight-defect estimate cannot be computed. The missing scores bound what it could have been: if every missing A run had succeeded and every missing B run had failed, the difference over the full schedule would be -25.0 percentage points; the reverse gives $+29.2$ percentage points. These data therefore do not show whether the instruction changes detection.

TABLE I
DETECTION PER RUN (THREE RUNS PER PROMPT; A DASH IS A MISSING SCORE) AND OUTCOME OF EACH SCORED SUITE: D DETECTED, R REACHED THE TRIGGERING PATH WITHOUT DETECTING, T COMBINED THE REQUIREMENTS WITHOUT REACHING THE PATH, N DID NOT COMBINE THEM.

Project/PR	Interaction	A	B	D	R	T	N
Celery 2598	JSON, task errors	0 0 0	0 0 0	0	0	6	0
Rails 18349	Keyless rollback	– – –	– – –				
Celery 2607	Eager retry	1 1 1	1 1 1	6	0	0	0
Spring 1526	Readiness, flush	0 0 0	0 0 0	0	0	3	3
Django 4150	Migration perms.	– – –	– – –				
Celery 2691	JSON, chord fail.	0 0 0	0 0 0	0	0	2	4
Flask 2254	Session teardown	0 0 0	0 0 0	0	1	5	0
Rails 18434	Route constraints	0 0 0	0 – 0	0	0	5	0
Scored suites				6	1	21	7

TABLE II
GENERATED TESTS BY PROMPT. THE 23 TESTS THAT COULD NOT RUN IN THE EVALUATION ENVIRONMENT HAVE UNKNOWN FLAGS.

Tests	A	B	Both
Written	61	70	131
Ran	51	57	108
Ran with a valid oracle	48	54	102
Combined both obligations	16	24	40
Reached the triggering path	3	4	7
Detected the defect	3	3	6

B. Combining, reaching, detecting

Table II follows the generated tests through the chain. Of 131 tests, 108 ran and 102 had a valid oracle. Forty of those combined both obligations. Seven reached the triggering path. Six detected the defect.

The large drop is between combining and reaching: 33 of the 40 combining tests never executed the faulty path. The step after it barely loses anything, since six of the seven tests that reached the path detected the defect. In this sample the agent's difficulty was neither choosing the combination nor writing the assertion. It was building a fixture in which the combination runs through the code that holds the defect.

The same pattern holds per suite and per defect (Table I, right). Of the 29 scored suites without a detection, 21 combined the requirements without reaching the path, seven did not combine them, and one reached the path without a qualifying assertion failure. Every one of the five undetected defects with scores has suites of the first kind: all six for Celery 2598, five of six for Flask 2254, all five for Rails 18434, three of six for Spring 1526 and two of six for Celery 2691.

An example. Celery 2691 concerns a chord, a group of tasks followed by a callback, whose header task fails while results are serialized as JSON. One generated test supplied a failing header, checked that the callback was not executed and checked for a `ChordError`. Those are the right public behaviors. The test used an in-memory result backend, which never enters the message-broker polling path where the historical defect lost the underlying error. It passed before and after the repair. In Flask 2254 a test did reach session opening and

raised an error before the repair, but an uncaught error is not an assertion failure, so it was not counted as a detection.

Effect of the instruction. The instruction changed what the tests combined. Prompt B produced 24 combining tests against 16 for A. In the 17 pairs with both runs scored, both suites combined the requirements in 11, only B did in five, and neither did in one. Three of the five B-only pairs belong to Spring 1526 and two to Celery 2691. None of them detected its defect, because none reached the path. Judged by whether tests exercise requirement combinations, the instruction works. Judged by detection, it shows no effect here.

V. Discussion

A passing test that combines two requirements looks like evidence that the combination works. These results show how weak that evidence can be when the fixture is chosen for convenience. An in-memory substitute keeps the public interface and removes the path on which the defect lives. The test then checks the right behavior of the wrong implementation.

This suggests a concrete check for a test generator. After choosing a combination and an assertion, it should trace whether the fixture can execute the implementation path responsible for the asserted behavior, and prefer a fixture that can. Whether giving an agent this check raises detection is a question for a new experiment. The measurement used here, which records combining, reaching and detecting separately, is what makes that experiment readable: a method that scored only whether tests combine requirements would have credited the instruction with an improvement that never reached detection.

Limits. Eight defects were studied and three come from Celery. Thirteen of 48 scores are missing, concentrated in Django and Rails, so the comparison of the two prompts is open in both directions. All seven suites blocked by the environment had passed the generation audit; counting them as failures would confuse the evaluator's environment with the agent's ability. One investigator qualified all defects with AI assistance, so an error in reading a requirement could recur across defects. Whether a test combined the requirements or reached the path is a judgment from the fixture and the execution record. The two graders' first-pass entries for these fields differed on 101 of 131 tests, counting unknown against explicit values, and the breakdown reported here is the adjudicated one. The historical code may have appeared in the model's training data. One model, one reasoning setting and a workflow without execution feedback limit what carries over to agents that can run their tests.

VI. Conclusion

For defects that need two requirements at once, the coding agent we studied usually thought of the combination and seldom reached the code that fails. Asking it to combine requirements gave more combining tests and the same detections. The next step is to give the generator a reachability check on its own fixtures and to measure detection again on a larger set of defects, with the evaluation environments validated first.

Data and code. The archive published with this paper (doi:10.5281/zenodo.24482952) holds the run and test score ledgers, a case index with the revision identifiers, and scripts that reproduce every table, the paired comparison and the bounds without model access. The qualification harnesses, the submitted test bodies and the raw grading and execution records are kept in the study archive and are needed for regrading or re-execution.

Acknowledgment

AI-based tools assisted with the programming and the writing of this paper.

References

[1] J. M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992. doi:10.1109/32.153381.

[2] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Practical Combinatorial Testing*. NIST Special Publication 800-142, 2010. doi:10.6028/NIST.SP.800-142.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi:10.1109/TSE.2014.2372785.

[4] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *ISSTA*, 2014. Author copy.

[5] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 2024. Author preprint.

[6] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. *ICSE*, 2023. Author preprint.

[7] N. Mündler, M. N. Mueller, J. He, and M. Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37, 2024. Proceedings.

[8] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. To mock or not to mock? An empirical study on mocking practices. *MSR*, 2017, pp. 402–412. doi:10.1109/MSR.2017.61.