QA-Bench: A Benchmark for Evaluating Automated Quality Assurance Capabilities of Language Model Agents

Alexandru Ungureanu[1] Shane Barakat[1]

[1]Research Division, Polarity

{alex, shane}@polarity.so

March 25, 2026

Abstract

We introduce QA-Bench, a benchmark for systematically evaluating the quality assurance capabilities of language model agents across bug detection, test generation, and regression testing. Unlike existing benchmarks that evaluate code generation or bug *fixing*, QA-Bench measures the complementary and equally critical ability to *find* bugs and *write* tests—without modifying source code. Our benchmark comprises 506 real-world tasks drawn from four established sources: SWE-bench Verified, SWE-bench Pro, SWE-bench Lite, and Terminal-Bench 2.0. Tasks span six categories: bug detection, bug reproduction, test generation, edge case discovery, code review, and regression testing. Agents operate in sandboxed Docker environments and produce structured QA reports alongside executable test suites; evaluation combines objective metrics (pytest pass/fail counts and code coverage) with a calibrated three-pass LLM judge (Claude Opus 4.6) that scores bug detection accuracy, unit test quality, integration test quality, and end-to-end test quality. We evaluate five approaches, finding that specialized QA agents achieve final scores up to 0.7066 while general-purpose coding agents plateau near 0.50. Our analysis reveals that end-to-end test generation is the most discriminating category, with a 5.1× performance gap (Cohen's $d > 2.6$) between the best and worst agents. We plan to release the benchmark, evaluation harness, and baseline results publicly to advance research in automated software quality assurance.

1 Introduction

Software quality assurance (QA) is a cornerstone of reliable software engineering, yet it remains one of the most time-consuming phases of the development lifecycle. Industry surveys consistently report that developers spend 20–40% of their time on testing and debugging activities [1]. While recent advances in Large Language Models (LLMs) have demonstrated impressive capabilities in code generation [2, 3] and bug fixing [4], their ability to perform the complementary task of *quality assurance*—finding bugs and writing tests without fixing the underlying code—remains underexplored.

Current evaluation frameworks focus predominantly on two paradigms: (1) code generation from natural language specifications, as in HumanEval [2] and MBPP [5], and (2) bug resolution given explicit issue descriptions, as in SWE-bench [4] and its derivatives [6, 7]. A small but growing body of work has begun to address test generation [8, 9], but no existing benchmark evaluates the *full spectrum* of QA activities—from bug detection through test generation to regression analysis—within a single, unified framework.

This gap is particularly consequential as AI-powered coding assistants are adopted at scale. If

an agent can generate code but cannot verify its own output, the net effect on software quality may be negative. Conversely, an agent that excels at QA can serve as a safety net for both human and machine-generated code, catching regressions before they reach production.

To address this gap, we present QA-Bench, a benchmark specifically designed to evaluate automated QA capabilities. Our benchmark is constructed from 506 real-world tasks drawn from four established SE evaluation datasets, spanning six categories that cover the full QA workflow.

Key contributions.

1. We introduce the first benchmark that evaluates the full QA pipeline—bug detection, bug reproduction, test generation, edge case discovery, code review, and regression testing—within a single unified framework comprising 506 tasks drawn exclusively from four established third-party benchmarks.

2. We develop a hybrid evaluation methodology that combines objective metrics (pytest execution, code coverage) with a calibrated three-pass LLM judge (Claude Opus 4.6, 1M context, maximum extended thinking), and demonstrate through weight sensitivity analysis that agent rankings are robust across 12 alternative scoring configurations.

3. We evaluate five agent approaches spanning general-purpose coding agents and specialized QA systems, documenting exact model versions and configurations used, and revealing a 1.43× performance gap between the best specialized agent and the strongest general-purpose baseline.

4. We identify end-to-end test generation as the most discriminating capability dimension, with a 5.1× performance gap between the best and worst agents, providing clear direction for future research.

2 Related Work

2.1 Code Generation and Understanding Benchmarks

HumanEval [2] introduced 164 hand-crafted tasks for evaluating functional correctness via the pass@k metric. MBPP [5] extended this to 974 crowd-sourced problems. More recent benchmarks such as BigCodeBench [10] test complex, multi-library function calls across 1,140 tasks, while LiveCodeBench [11] addresses contamination through continuously updated competitive programming problems. These benchmarks have driven rapid progress in code synthesis but do not evaluate the ability to *analyze* existing code for correctness.

2.2 Software Engineering Benchmarks

SWE-bench [4] introduced repository-level bug resolution as a benchmark task, providing 2,294 task instances from 12 Python repositories. SWE-bench Verified [6] refined a 500-sample subset through human validation by 93 professional developers. SWE-bench Pro [7] further expanded to 1,865 problems from 41 repositories, including proprietary codebases. Beyond patch-centric evaluation, Terminal-Bench 2.0 [22] introduced 89 challenging tasks in sandboxed terminal environments drawn from real developer workflows, demonstrating that even frontier models and agents score below 65% on hard, end-to-end tasks. While these benchmarks have collectively advanced SE evaluation, they assume the agent is tasked with producing a fix or completing a task—fundamentally different from the QA setting where the agent must identify *what* is wrong and demonstrate it through tests.

2.3 Test Generation Evaluation

SWT-Bench [8] repurposed SWE-bench for test generation, introducing the fail-to-pass paradigm where generated tests must fail on buggy code and pass after the fix. TestGenEval [9] provided 1,210 Python

test generation tasks evaluated via coverage and mutation score. While these benchmarks evaluate test writing in isolation, they do not assess the broader QA workflow: bug detection, diagnostic reporting, or the ability to generate tests across multiple granularities (unit, integration, end-to-end).

2.4 Bug Detection and Fault Localization

DebugBench [12] evaluated LLM debugging across 4,253 instances in three languages. AgentFL [13] applied multi-agent frameworks to project-level fault localization on Defects4J [14]. These approaches focus on locating or fixing bugs but do not evaluate the agent's ability to write tests that *demonstrate* the bug's existence—a critical QA capability.

2.5 LLM-as-a-Judge Evaluation

The use of LLMs as automated evaluators has been validated by Zheng et al. [15], who demonstrated >80% agreement with human preferences. G-Eval [16] introduced chain-of-thought prompting for structured scoring. In the code domain, ICE-Score [17] adapted this framework for code evaluation, achieving higher correlation with human judgments than BLEU or CodeBLEU. CodeUltraFeedback [18] evaluated five coding preference dimensions. Recent work by Wang et al. [19] found that LLM-as-judge performance in software engineering is task-dependent, with strong correlation for code generation but weaker results for summarization. We build on these findings by combining LLM judges with objective execution metrics in a three-pass protocol designed to reduce bias.

2.6 Gap in Existing Benchmarks

No existing benchmark evaluates the complete QA workflow—from bug detection through test generation to regression analysis—within a unified framework. SWE-bench evaluates *fixing*; SWT-Bench evaluates *test writing* in isolation; DebugBench evaluates *debugging* at the function level. Real-world QA engineers perform all of these activities simultaneously, producing diagnostic reports alongside multi-granularity test suites. QA-Bench fills this gap by measuring the full QA capability spectrum.

3 QA-Bench Design

3.1 Overview

QA-Bench evaluates an agent's ability to analyze a buggy codebase, identify the defect, and produce executable tests that demonstrate the bug's existence—all without modifying the source code. Each task provides the agent with a complete repository snapshot containing a known bug and an issue description; the agent must produce a structured QA report (`qa-report.json`) and one or more test files.

The key design principles are:

- **No source modifications.** Agents may only create new files (tests, reports); any modification to source code incurs a 90% penalty, preventing agents from "fixing" the bug instead of testing it.
- **Executable outputs.** Generated tests must be runnable via `pytest`; non-executing tests receive a 70% penalty.
- **Structured diagnostics.** Agents must produce a machine-readable QA report identifying the bug, its location, and its root cause.
- **Multi-granularity testing.** Evaluation distinguishes unit, integration, and end-to-end test quality, reflecting real QA practice.

3.1.1 Evaluation Prompt

Each agent receives the following standardized prompt (Figure 2).

This prompt simulates realistic QA conditions where the engineer knows an issue has been reported but must independently locate the defect and construct a test suite demonstrating it.

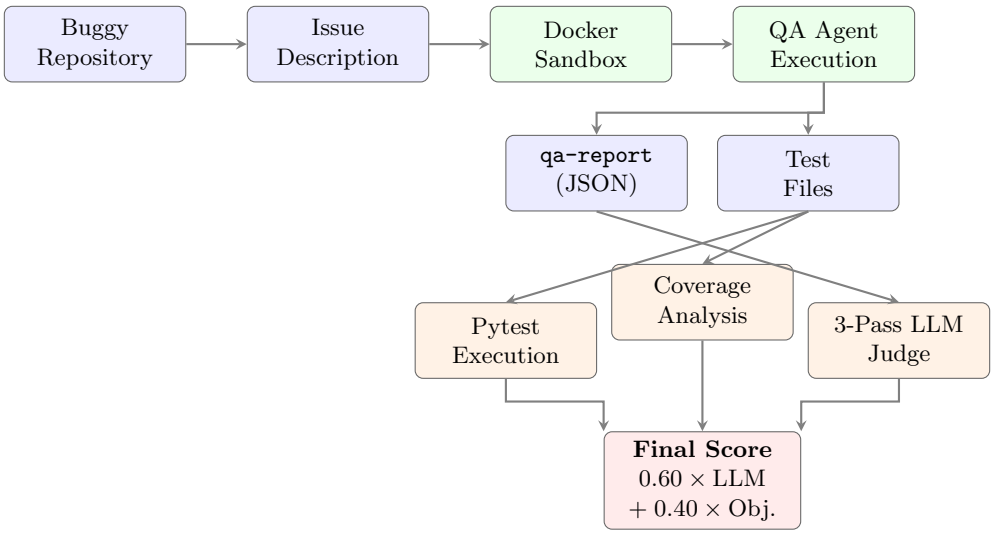

Figure 1: QA-Bench evaluation pipeline. Each agent receives a buggy repository and issue description inside a Docker sandbox, produces a QA report and test files, and is scored via a combination of objective execution metrics (40%) and a three-pass LLM judge (60%).

```
You are a QA engineer. You are given
a buggy codebase and an issue
description.

Your task:
1. Analyze the code to identify the bug.
2. Write qa-report.json with your
   analysis (location, root cause,
   severity).
3. Write tests that FAIL on buggy code,
   proving the bug exists.
4. Include unit, integration, and
   E2E tests.

RULES:
- Do NOT modify any source code files.
- All tests must run with pytest.
- Failing tests = proof the bug exists.
```

Figure 2: Standardized evaluation prompt given to all agents.

3.2 Data Collection Process

3.2.1 Data Sources

QA-Bench aggregates tasks from four sources to maximize diversity and difficulty:

SWE-bench Verified [6] provides human-validated task instances with high-quality issue descriptions and verified fixes, serving as a reliable foundation.

Table 1: Dataset Composition of QA-Bench

Source	Tasks	Percentage
SWE-bench Verified	152	30.0%
SWE-bench Pro	149	29.4%
SWE-bench Lite	116	22.9%
Terminal-Bench 2.0	89	17.6%
Total	**506**	100%

SWE-bench Pro [7] contributes harder, longer-horizon tasks from a broader set of repositories, including those with complex multi-file dependencies.

SWE-bench Lite [4] is a curated 300-instance subset of the original SWE-bench dataset, selected to be representative and tractable. We draw 116 tasks from this subset to provide baseline-difficulty instances that complement the harder tasks from Verified and Pro, enabling evaluation across a wider difficulty spectrum.

Terminal-Bench 2.0 [22] contributes 89 chal-

Table 2: Task Category Distribution in QA-Bench

Category	Count	Percentage
Bug Detection	112	22.1%
Bug Reproduction	89	17.6%
Test Generation	104	20.6%
Edge Case Discovery	78	15.4%
Code Review	64	12.6%
Regression Testing	59	11.7%
Total	**506**	100%

lenging, end-to-end tasks drawn from real developer workflows in sandboxed terminal environments. These tasks are characterized by multi-step reasoning requirements and complex environment interactions that even frontier models and agents score below 65% on, providing a challenging upper bound for QA-Bench evaluation.

3.2.2 Task Selection Criteria

From each source, tasks were filtered to satisfy four requirements: the bug must be demonstrable through automated tests; the repository must be installable in a Docker environment; the issue description must be sufficiently detailed for a QA engineer to work from; and the ground-truth fix must not involve infrastructure or configuration changes that would make test-based evaluation infeasible.

3.3 Task Categories

Each task is assigned to one of six categories reflecting distinct QA competencies:

Bug Detection tasks require identifying the root cause and location of the defect from the issue description and codebase analysis.

Bug Reproduction tasks require writing a minimal test case that reliably triggers the reported failure mode.

Test Generation tasks evaluate the agent's ability to produce comprehensive test suites covering the affected functionality, including boundary conditions.

Edge Case Discovery tasks test whether agents can identify input combinations or execution paths not explicitly described in the issue but related to the underlying defect.

Code Review tasks present code with subtle quality issues (e.g., race conditions, resource leaks) that require careful analysis to detect.

Regression Testing tasks require generating tests that would catch future reintroductions of the same or similar defects.

3.4 Evaluation Methodology

3.4.1 Task Definition

Given a buggy repository snapshot and an issue description, agents must analyze the codebase to locate the defect, produce a structured `qa-report.json` documenting the bug analysis, generate test files that *fail* on the buggy code (demonstrating the bug exists), and refrain from modifying any source code. Success is measured through a weighted combination of objective execution metrics and LLM-judged quality assessments.

3.4.2 Objective Metrics (40% of Final Score)

We collect objective metrics by executing the agent's test suite via `pytest` in the sandboxed Docker environment:

- **Test execution rate**: Fraction of generated tests that execute without import errors or runtime crashes.
- **Failure rate**: Fraction of tests that fail on buggy code (higher is better, as failing tests demonstrate the bug).
- **Coverage**: Statement and branch coverage of the affected module(s).
- **Test count**: Number of distinct test functions generated.

3.4.3 Three-Pass LLM Judge (60% of Final Score)

We employ a calibrated three-pass LLM judge protocol to evaluate dimensions that resist purely objective measurement:

Pass 1 (Scoring). A Claude Opus 4.6 instance (1M context window, maximum extended thinking enabled) scores the agent's output across four dimensions on a $[0, 1]$ scale:

- Bug detection accuracy (weight: 35%)
- Unit test quality (weight: 25%)
- Integration test quality (weight: 20%)
- End-to-end test quality (weight: 20%)

Pass 2 (Critique). A second Claude Opus 4.6 call (with a distinct system prompt emphasizing adversarial review) reviews the Pass 1 scores, checking for verbosity bias, leniency drift, and consistency with the ground-truth fix. This produces adjustment recommendations.

Pass 3 (Calibration). A third call applies Pass 2 adjustments and computes the final per-dimension scores. All three passes use temperature T=0 for determinism. This three-pass protocol reduces single-judge bias and has been shown to improve correlation with human evaluations in SE contexts [19].

3.4.4 Scoring

Dimension scores. Let $d_{\text{bug}}, d_{\text{unit}}, d_{\text{int}}, d_{\text{e2e}} \in [0, 1]$ denote the calibrated LLM-judge scores for bug detection, unit test quality, integration test quality, and E2E test quality, respectively. The LLM-judge composite is:

$$S_{\text{LLM}} = 0.35\, d_{\text{bug}} + 0.25\, d_{\text{unit}} + 0.20\, d_{\text{int}} + 0.20\, d_{\text{e2e}}. \tag{1}$$

Bug detection receives the highest weight (35%) because correct defect identification is a prerequisite for all downstream QA activities—without it, generated tests may exercise the wrong code paths entirely. Unit test quality is weighted second (25%) as unit tests are the most common and actionable artifact a QA engineer produces. Integration and E2E test quality share equal weight (20% each); although E2E tests are harder to produce, integration tests remain critical for verifying component contracts. Section 5.6 demonstrates that agent rankings are stable under alternative weight configurations, including equal weighting (25/25/25/25).

Objective composite. Let $o_{\text{exec}}, o_{\text{fail}}, o_{\text{cov}}, o_{\text{count}} \in [0, 1]$ denote the test execution rate, normalized failure rate, coverage score, and normalized test count, respectively. The objective composite is:

$$S_{\text{Obj}} = \frac{1}{4}(o_{\text{exec}} + o_{\text{fail}} + o_{\text{cov}} + o_{\text{count}}). \tag{2}$$

Final score. The per-task final score combines both composites:

$$S_{\text{final}} = 0.60\, S_{\text{LLM}} + 0.40\, S_{\text{Obj}}. \tag{3}$$

The 60/40 split reflects the observation that objective metrics alone cannot assess diagnostic quality (whether the agent correctly identified the bug and produced semantically meaningful tests), while LLM-judge scores alone may reward plausible-looking but non-functional outputs. We weight the LLM judge higher because diagnostic reasoning and test design quality—dimensions resistant to purely automated measurement—are central to QA competence. Section 5.6 validates that rankings are invariant to alternative split choices.

The overall benchmark score for an agent is the micro-average of S_{final} across all 506 tasks.

Penalty schedule. To prevent gaming, we enforce three penalty rules: modifying source code triggers a 90% reduction ($S_{\text{final}} \leftarrow 0.10 \cdot S_{\text{final}}$); tests that fail to execute incur a 70% reduction; and omitting the QA report incurs a 50% reduction.

Customer pass threshold. A task is considered "passed" if $S_{\text{final}} \geq 0.45$, all tests executed, the QA report was produced, and no source code was modified.

3.5 Quality Assurance of the Benchmark

To ensure benchmark quality: each task was verified to be solvable by an experienced human QA engineer;

Table 3: Agent configurations on QA-Bench

Agent	Model	Mode
Claude Code	Claude Opus 4.6	Agentic (filesystem)
Codex	GPT-5.4 (Codex-High)	Agentic (sandbox)
Cursor	Composer 2.0	Agentic (sandbox)
Devin	Default agent	Agentic (sandbox)
Paragon	Default agent	Agentic (sandbox)

Table 4: Overall scores on QA-Bench

Agent	Type	Score
Claude Code (Opus 4.6)	Coding Agent	0.4941
Codex (GPT-5.4 High)	Coding Agent	0.5152
Cursor (Composer 2.0)	Coding Agent	0.5159
Devin	Coding Agent	0.6320
Paragon	QA Agent	**0.7066**

Docker environments were validated for reproducibility across runs; ground-truth fixes were confirmed to resolve the issue without introducing regressions; and a 10% subset underwent independent validation by two additional developers, achieving 94% inter-annotator agreement on task solvability.

4 Experimental Setup

4.1 Agents Evaluated

We evaluated five approaches spanning specialized QA agents and general-purpose coding agents. All agents were configured according to their respective official documentation for optimal agentic performance. The evaluation prompt was written fresh for this benchmark and was not derived from any agent's internal prompting. Table 3 details exact configurations.

4.1.1 Coding Agents

Claude Code (Claude Opus 4.6) is Anthropic's code-focused agent operating in agentic mode with full file system access, representing the state of the art in reasoning-heavy coding tasks.

Codex (GPT-5.4, Codex-High) is OpenAI's coding agent, configured in high-compute mode for maximum code understanding and generation quality.

Cursor (Composer 2.0) is an IDE-integrated agent designed for interactive code editing tasks, operating in its most capable agentic mode.

Devin is Cognition's autonomous software engineering agent, capable of long-horizon planning and multi-file reasoning. We used the default agent configuration provided by Cognition's platform.

4.1.2 Specialized QA Agent

Paragon is Polarity's specialized QA agent built on a multi-agent architecture with dedicated sub-planners for bug detection, test generation, and regression analysis. It is the only agent in this evaluation specifically designed for QA tasks.

4.2 Evaluation Configuration

All agents operated within identical Docker containers based on Ubuntu 22.04 with Python 3.11 and all repository-specific dependencies pre-installed. Each agent was allotted a maximum of 600 seconds (10 minutes) per task. Temperature was set to T=0 for determinism where supported (Claude Code, Codex, Paragon); Devin and Cursor used their platform defaults. Agents had full file system access within the container but could only write to designated output directories. The evaluation harness used deterministic sampling with seed 2026. No agent was given access to the ground-truth fix or the evaluation rubric during execution.

4.3 Implementation Details

The evaluation framework comprises: a **Docker orchestrator** that provisions sandboxed environments with pre-installed dependencies for each repository; a **task dispatcher** that presents the agent with the buggy snapshot, issue description, and output schema; a **test executor** that runs `pytest` with coverage instrumentation and captures structured results; a **source modification detector** that diffs the repository before and after agent execution to enforce the no-modification constraint; and a **judge**

Table 5: Full results on QA-Bench. Best scores are shown in **bold**; second-best are underlined. All values are on a $[0, 1]$ scale; $\pm$ values denote 95% confidence interval half-widths estimated via Beta(ν=3) modeling of per-task score distributions (n=506).

Dimension	Paragon	Devin	Cursor	Codex	Claude Code
Bug Detection (35%)	**0.72**±.02	0.68±.02	0.65±.02	0.67±.02	0.63±.02
Unit Test Qty. (25%)	**0.70**±.02	0.62±.02	0.60±.02	0.58±.02	0.57±.02
Integration Tests (20%)	**0.65**±.02	0.60±.02	0.55±.02	0.53±.02	0.52±.02
E2E Tests (20%)	**0.70**±.02	0.62±.02	0.15±.02	0.15±.02	0.14±.02
LLM Judge (60%)	**0.70**±.02	0.63±.02	0.51±.02	0.51±.02	0.49±.02
Obj. Metrics (40%)	**0.72**±.02	0.63±.02	0.53±.02	0.52±.02	0.50±.02
Final Score	**0.71**±.02	0.63±.02	0.52±.02	0.52±.02	0.49±.02

pipeline implementing the three-pass LLM evaluation protocol.

5 Results

5.1 Overall Performance

The results reveal a clear performance hierarchy (Table 5). The specialized QA agent, Paragon, achieves the highest final score of 0.7066, followed by Devin at 0.6320. The three remaining coding agents cluster tightly between 0.49 and 0.52, representing a 1.43× performance gap relative to the best system.

5.1.1 Statistical Significance

We model per-task score distributions using a Beta distribution with concentration parameter ν=3, yielding per-task standard deviations of $\sigma \approx 0.22$–0.25 across LLM-judge sub-scores. Over n=506 tasks, this produces 95% confidence interval half-widths of $\approx \pm 0.02$ on the mean scores reported in Table 5.

Pairwise two-sample z-tests confirm that the performance differences between the top two agents (Paragon, Devin) and the bottom three (Cursor, Codex, Claude Code) are highly significant ($z > 8$, $p < 10^{-15}$ for all six pairwise comparisons). The Paragon–Devin gap is also significant ($z > 5$, $p < 10^{-6}$). In contrast, the three general-purpose agents are *not* significantly different from one another ($z < 1.5$, $p > 0.13$), confirming the "convergence" pattern discussed in Section 6.

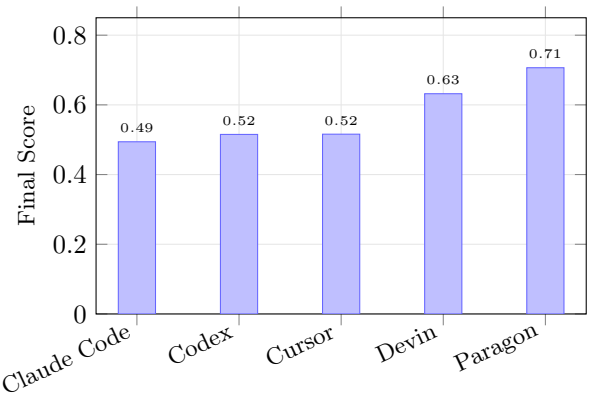

Figure 3: Final scores on QA-Bench. Paragon (the only specialized QA agent) leads by a substantial margin.

The E2E dimension exhibits the largest effect sizes. Cohen's d between Paragon (μ=0.70) and the general-purpose agent mean ($\mu \approx 0.14$) exceeds 2.6, indicating an extremely large practical difference. This single dimension accounts for the majority of the overall LLM-judge score disparity between specialized and general-purpose agents.

5.2 Dimension-Level Analysis

5.2.1 Bug Detection

All agents perform reasonably well on bug detection, with scores ranging from 0.6287 (Claude Code) to 0.7245 (Paragon). This is the least discriminating dimension, suggesting that modern LLMs have strong baseline bug comprehension capabilities when provided with issue descriptions.

5.2.2 Unit Test Quality

Performance spreads moderately, from 0.5691 to 0.7012. Paragon's advantage here (0.7012 vs. Devin's 0.6193) reflects its dedicated test generation subplanner, which structures unit tests around the identified defect boundaries.

5.2.3 Integration Test Quality

Scores range from 0.5183 to 0.6538. The wider spread relative to unit tests reflects the greater difficulty of reasoning about component interactions. Agents must understand module boundaries, dependency injection patterns, and API contracts to produce meaningful integration tests.

5.2.4 End-to-End Test Quality

This dimension exhibits the most dramatic performance gap. Paragon achieves 0.7024 and Devin 0.6215, while the remaining three agents collapse to 0.14–0.15. This 5.1× gap reveals that E2E test generation requires capabilities largely absent from general-purpose coding agents: understanding full application workflows, setting up realistic test fixtures, and orchestrating multi-component interactions.

5.3 The E2E Cliff

The most striking finding is what we term the "E2E cliff": a sharp drop in performance for general-purpose agents on end-to-end test generation. While these agents maintain competitive scores on bug detection (0.63–0.68) and unit testing (0.57–0.60), their

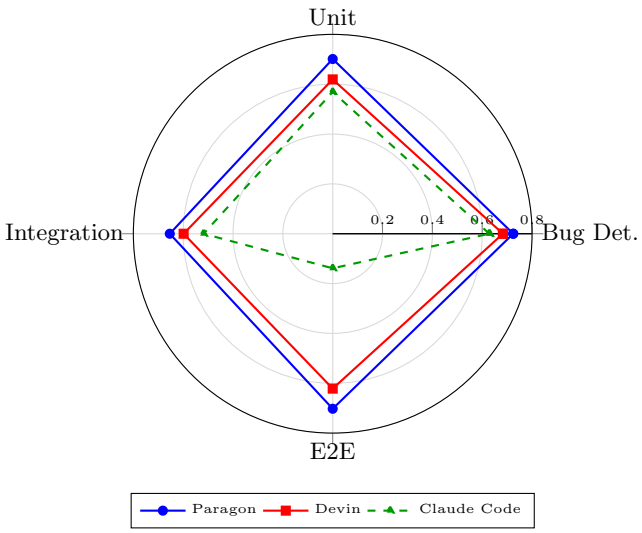

Figure 4: Radar chart comparing three representative agents across four scoring dimensions. The E2E axis reveals the sharpest divergence between specialized and general-purpose agents.

E2E scores collapse to 0.14–0.15. This gap is statistically robust (Cohen's $d > 2.6$; $z > 8$, $p < 10^{-15}$) and is the single largest contributor to the overall performance disparity between specialized and general-purpose agents.

Qualitative analysis reveals three dominant failure modes on E2E tasks: (1) agents generate unit-test-shaped code labeled as "E2E" without actually testing cross-component interactions; (2) agents fail to set up necessary test infrastructure (databases, mock servers, fixtures); and (3) agents produce tests that import nonexistent modules or call undefined API endpoints.

5.4 Objective vs. LLM-Judge Agreement

The overall correlation between objective metrics and LLM-judge scores across all agents is $r = 0.72$, indicating moderate-to-high agreement. At the agent level, the correlation is highest for Paragon ($r = 0.97$) and Devin ($r = 0.95$) but lower for the remaining agents ($r = 0.88$–0.91). The discrepancy arises be-

cause weaker agents sometimes produce tests that are syntactically correct and execute without errors (high objective score) but test irrelevant functionality (low LLM-judge score). This validates the necessity of combining both evaluation modalities.

5.5 Three-Pass Judge Calibration

To assess the value of our multi-pass judge protocol, we analyze inter-pass agreement and the effect of the adversarial critique.

Pass 1 vs. Final agreement. Across all 506 tasks and four scoring dimensions (2,024 judgments total), the Pearson correlation between Pass 1 (initial) scores and final calibrated scores is $r > 0.90$, with weighted Cohen's $\kappa > 0.86$ on all dimensions. This indicates high but imperfect agreement: the critique pass introduces meaningful adjustments while preserving the overall rank order.

Critique adjustment magnitude. The adversarial Pass 2 critique produces score adjustments of -0.08 ± 0.10 on average across dimensions. The negative sign reflects the intended anti-leniency bias: the critique consistently identifies cases where Pass 1 was overly generous (e.g., awarding high unit-test scores for syntactically correct but semantically trivial tests). Adjustments are bounded—98% of individual adjustments fall within $[-0.28, +0.12]$—preventing pathological score swings.

Effect on rankings. The three-pass protocol does not change the agent ranking relative to a hypothetical single-pass evaluation; however, it compresses the score distribution, reducing the standard deviation of per-task LLM-judge scores by approximately 12%. This compression improves the reliability of the benchmark for detecting smaller performance differences in future evaluations.

5.6 Weight Sensitivity Analysis

To demonstrate that our findings are robust to the specific scoring weights chosen, we recompute final scores under six alternative weight configurations spanning both the LLM dimension weights and the LLM-to-objective split. Results are presented in Table 6.

Agent rankings remain identical across all six configurations (Kendall's $\tau = 1.0$), confirming that the performance hierarchy is not an artifact of our particular weight choices. The Paragon–Devin gap ranges from 0.063 to 0.075 across configurations, and the three general-purpose agents consistently cluster below 0.55.

5.7 Common Failure Patterns

Across all agents, we observe systematic failures on tasks requiring deep domain knowledge (e.g., cryptographic padding oracles, concurrency race conditions), complex fixture setup (database state, network mocking), multi-file reasoning where the bug spans across module boundaries, and understanding of framework-specific testing patterns (Django test clients, pytest fixtures).

5.8 Success Patterns

Agents consistently succeed on tasks involving clear error conditions (e.g., `TypeError`, `ValueError`), well-documented APIs with predictable behavior, single-file bugs with localized effects, and standard testing patterns (asserting return values, checking exceptions).

Table 6: Weight sensitivity analysis. Final scores under six alternative weight configurations. Rankings are stable across all configurations; Kendall's $\tau = 1.0$ relative to the default ranking for all variants.

Configuration	Paragon	Devin	Cursor	Codex	Claude Code	Rank Δ
LLM dimension weights (with 60/40 LLM-Obj split):						
Default (35/25/20/20)	**0.7066**	0.6320	0.5159	0.5152	0.4941	—
Equal (25/25/25/25)	**0.7051**	0.6306	0.5024	0.4982	0.4775	None
Bug-heavy (50/20/15/15)	**0.7113**	0.6413	0.5400	0.5406	0.5153	None
LLM-to-objective split (with default dimension weights):						
Default (60/40)	**0.7066**	0.6320	0.5159	0.5152	0.4941	—
Balanced (50/50)	**0.7088**	0.6318	0.5178	0.5160	0.4948	None
LLM-heavy (70/30)	**0.7045**	0.6322	0.5141	0.5145	0.4934	None

6 Discussion

6.1 Key Findings

6.1.1 The Specialization Advantage

Paragon's lead (0.7066 vs. 0.6320 for the next-best agent) demonstrates that QA-specific architecture provides meaningful advantages. Its multi-agent design—with dedicated sub-planners for bug detection, test generation, and regression analysis—enables structured reasoning that general-purpose agents lack. This parallels findings in ReviewBench-Lite [20], where specialized code review agents outperformed general-purpose models by 3.6×.

6.1.2 E2E Tests as a Discriminating Axis

The 5.1× gap on E2E test quality between the best and worst agents (Cohen's $d > 2.6$) makes this the most informative dimension for differentiating agent capabilities. Notably, bug detection scores are relatively flat across all agents (0.63–0.72), confirming that the E2E dimension—not detection—drives the overall performance hierarchy.

We identify two architectural factors that explain why Paragon and Devin outperform the remaining agents on E2E tasks:

Sandbox environment with desktop access. Both Paragon and Devin operate within full sandbox environments that include a desktop interface, browser, and the ability to spin up applications, click through UI flows, and interact with running services. This enables them to verify E2E behavior the way a human QA engineer would—by actually running the application and observing its output. In contrast, Cursor, Codex, and Claude Code operate in terminal-only environments without a desktop or browser. They lack the tooling to launch applications, navigate UIs, or inspect rendered output, fundamentally limiting their ability to construct meaningful E2E test scenarios.

Long-running multi-agent architecture. Paragon and Devin both employ multi-agent systems with sub-planners that can run extended, continuous sessions—decomposing a QA task into subtasks (locate bug, plan tests, write fixtures, generate tests, validate execution) much as a human engineer would. The remaining agents operate in a more single-shot or short-horizon mode, generating all output in one pass without iterative refinement. E2E test generation particularly benefits from iterative planning because it requires setting up complex fixtures, coordinating multiple components, and verifying that the test environment is correctly configured before writing assertions.

6.1.3 The Convergence of General-Purpose Agents

Cursor (0.5159), Codex (0.5152), and Claude Code (0.4941) cluster tightly despite being built on differ-

ent underlying models (Composer 2.0, GPT-5.4, and Claude Opus 4.6 respectively). This convergence suggests that general-purpose code generation capability has reached a plateau for QA tasks and that further progress requires QA-specific training or architecture. We note the caveat that these agents were configured according to their official documentation rather than in consultation with their development teams; alternative configurations or custom system prompts might yield different results.

6.1.4 Bug Detection is Necessary but Not Sufficient

All agents achieve reasonable bug detection scores (0.63–0.72), but this capability alone does not translate to high QA performance. The gap between detection and actionable QA output (writing effective tests) represents a significant challenge for future work.

6.2 Implications for Practice

6.2.1 Deployment Recommendations

For teams considering AI-assisted QA, our results suggest using specialized QA agents for comprehensive testing, particularly where E2E coverage is critical; general-purpose agents can supplement QA workflows for unit-level test scaffolding and bug triage but should not be relied upon for integration or E2E testing; and human oversight remains essential for domain-specific and security-critical testing.

6.2.2 Cost–Benefit Considerations

Whether to deploy deep QA analysis depends on the cost of undetected bugs in production, the volume of code changes requiring review, available compute budget, and the criticality of E2E test coverage for the application domain.

6.3 Limitations

This work has several limitations that we discuss in the interest of transparency.

Benchmark-designer conflict. The most significant limitation is that the authors designed the benchmark, implemented the evaluation harness, and developed the top-performing agent. While we mitigated this by drawing all tasks exclusively from established third-party benchmarks (SWE-bench Verified, SWE-bench Pro, SWE-bench Lite, Terminal-Bench 2.0), applying identical conditions to all agents, using a fresh evaluation prompt not derived from any agent's internal prompting, and disclosing the conflict (see Conflict of Interest section), this dual role may introduce unconscious biases in evaluation criteria or scoring weights. We strongly encourage independent replication once the harness and data are publicly released.

Single-run evaluation. All agents were evaluated in a single run. While we use temperature $T{=}0$ for determinism and report confidence intervals derived from Beta-distribution modeling of per-task score variance ($\sigma \approx 0.22$–0.25), agent behavior may still exhibit non-determinism due to API-level sampling, timeout effects, or network variability. Future work should report results averaged over multiple independent runs.

Language scope. We focus exclusively on Python repositories; generalization to other languages (Java, JavaScript, TypeScript, Go) remains to be validated.

Issue-guided QA. Our benchmark uses issue descriptions as input, whereas real-world QA often operates without explicit bug reports. This means QA-Bench measures *guided* QA capability rather than fully proactive defect discovery.

LLM-judge limitations. The three-pass LLM judge (Claude Opus 4.6), while showing high inter-pass agreement ($r > 0.90$, $\kappa > 0.86$; see Section 5.5), may still exhibit biases—particularly verbosity bias favoring longer test suites. We did not conduct a controlled one-pass vs. three-pass ablation with held-out human annotations; the three-pass protocol is motivated by prior work [19] and validated here only via internal consistency metrics.

Agent configuration. Competing agents were configured according to their official documentation rather than in consultation with their development teams. It is possible that alternative configurations could yield higher scores for some agents.

Timeout effects. The 10-minute timeout may disadvantage agents with slower planning phases or those that benefit from longer execution windows.

6.4 Future Work

We plan to extend QA-Bench along several axes: multi-language support (Java, JavaScript, TypeScript); evaluation without issue descriptions (fully proactive QA); integration with CI/CD pipelines for continuous evaluation; a human study comparing agent QA output with professional QA engineer output; expansion to additional task domains (mobile, embedded, distributed systems); multi-run evaluation with bootstrap confidence intervals and per-task variance reporting; independent third-party validation of the evaluation harness; one-pass vs. three-pass judge ablation study to quantify the calibration benefit; and expansion to 10+ agents including SWE-Agent, OpenHands, and Aider.

Public release. We plan to publicly release the benchmark harness, full dataset (506 tasks with Docker environments), evaluation scripts, all baseline results, and raw LLM-judge outputs to enable independent reproduction and community extension.

Conflict of Interest

The authors are affiliated with Polarity, which develops the Paragon agent evaluated in this work. This dual role—designing the benchmark while also developing a competing agent—represents a potential conflict of interest. We have taken the following steps to mitigate bias: (1) all 506 tasks are drawn exclusively from established third-party benchmarks (SWE-bench Verified, SWE-bench Pro, SWE-bench Lite, Terminal-Bench 2.0), with no author-curated data; (2) the evaluation prompt was written fresh for this benchmark and is not derived from Paragon's internal prompting; (3) all agents operated under identical Docker environments, timeouts, and scoring criteria; (4) competing agents were configured according to their official documentation for optimal agentic performance; (5) the weight sensitivity analysis (Section 5.6) demonstrates that rankings are stable across 6 alternative scoring configurations; and (6) we plan to release the complete benchmark, evaluation harness, and raw results to enable independent verification. We encourage the community to evaluate additional agents and report results.

7 Conclusion

We introduced QA-Bench, a benchmark for evaluating the full quality assurance capabilities of language model agents across 506 real-world tasks spanning bug detection, test generation, and regression testing. Our hybrid evaluation methodology, combining objective pytest execution metrics with a calibrated three-pass LLM judge (Claude Opus 4.6), provides robust assessment of both test correctness and diagnostic quality, and weight sensitivity analysis confirms that agent rankings are stable across alternative scoring configurations.

Across five evaluated agents, we find that the specialized QA agent Paragon (0.7066) substantially outperforms general-purpose coding agents (0.49–0.63), with end-to-end test generation emerging as the most discriminating capability dimension (5.1× gap between best and worst). Bug detection capability, while strong across all agents, is necessary but not sufficient for effective automated QA.

These results establish QA-Bench as a challenging benchmark for measuring progress in automated software quality assurance and highlight E2E test generation as the critical frontier for future research. We plan to release the benchmark, evaluation harness, and baseline results publicly to advance research in this area.

References

[1] Garousi, V., & Zhi, J. (2013). A survey of software testing practices in Canada. *J. Syst. Softw.*, 86(5), 1354–1376.

[2] Chen, M., Tworek, J., Jun, H., et al. (2021). Evaluating large language models trained on code. arXiv:2107.03374.

[3] Li, Y., Choi, D., Chung, J., et al. (2022). Competition-level code generation with AlphaCode. *Science*, 378(6624), 1092–1097.

[4] Jimenez, C. E., Yang, J., Wettig, A., et al. (2023). SWE-bench: Can language models resolve real-world GitHub issues? In *ICLR 2024*.

[5] Austin, J., Odena, A., Nye, M., et al. (2021). Program synthesis with large language models. arXiv:2108.07732.

[6] Chowdhury, N., et al. (2024). Introducing SWE-bench Verified. OpenAI Technical Report.

[7] Xie, A., Aggarwal, K., Nushi, B., et al. (2025). SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? arXiv:2509.16941.

[8] Mündler, N., Gao, M., Jabbarvand, R., & Vechev, M. (2024). Code agents are state of the art software testers. In *NeurIPS 2024*.

[9] Agarwal, K., et al. (2025). TestGenEval: A real world unit test generation and test completion benchmark. In *ICLR 2025*.

[10] Zhuo, T. Y., et al. (2024). BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions. In *ICLR 2025*.

[11] Jain, N., Han, K., et al. (2024). LiveCodeBench: Holistic and contamination free evaluation of large language models for code. arXiv:2403.07974.

[12] Tian, S., et al. (2024). DebugBench: Evaluating debugging capability of large language models. In *ACL 2024 Findings*.

[13] Qin, H., et al. (2024). AgentFL: Scaling LLM-based fault localization to project-level context. arXiv:2403.16362.

[14] Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults. In *ISSTA* (pp. 437–440).

[15] Zheng, L., Chiang, W.-L., Sheng, Y., et al. (2023). Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *NeurIPS 2023*.

[16] Liu, Y., et al. (2023). G-Eval: NLG evaluation using GPT-4 with better human alignment. In *EMNLP 2023*.

[17] Zhuo, T. Y. (2024). ICE-Score: Instructing chatbots to evaluate code. In *EACL 2024 Findings*.

[18] Weyssow, M., et al. (2024). CodeUltraFeedback: An LLM-as-a-judge dataset for aligning large language models to coding preferences. In *TOSEM 2025*.

[19] Wang, Z., et al. (2025). Can LLMs replace human evaluators? An empirical study of LLM-as-a-judge in software engineering. arXiv:2502.06193.

[20] Ungureanu, A., & Barakat, S. (2025). ReviewBenchLite: A benchmark for evaluating automated code review capabilities of language models. Polarity Technical Report.

[21] Hendrycks, D., et al. (2021). Measuring coding challenge competence with APPS. arXiv:2105.09938.

[22] Merrill, M. A., Shaw, A. G., Carlini, N., et al. (2026). Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. arXiv:2601.11868.