FROM BENCHMARK SCORES TO DEPLOYMENT READINESS: A JOURNAL-SCALE EVALUATION FRAMEWORK FOR AUTONOMOUS SOFTWARE DEVELOPMENT AGENTS

Partha Sarathi Samal[1], Suresh Kumar Palus[2], and Sai Kiran Padmam[3]

[1]Independent Researcher, Connecticut, USA

[2]Independent Researcher, Pennsylvania, USA

[3]Independent Researcher, New Jersey, USA

Abstract

Autonomous software engineering agents are progressing from code assistants to systems that can plan and execute multi-step repository changes. This shift requires evaluation methods that move beyond one-shot pass rate reporting. Many current studies still provide limited visibility into repeatability, failure recovery, and operational efficiency under realistic constraints. This journal article presents an expanded DevAgentBench and DevAgentEval methodology designed for deployment-oriented assessment. The benchmark covers bug fixing, test generation, refactoring, code review assistance, and long-horizon feature work. We organize analysis into three metric layers: task-level success and correctness, robustness under perturbation, and business-aligned operational efficiency. We also formalize a nine-category failure-mode taxonomy linked to trace-level evidence and remediation guidance. Baseline experiments across agent patterns and model families show that rankings are sensitive to context reduction, tool-output noise, transient execution failures, and tighter resource budgets. These findings indicate that average success rates alone are insufficient for production decisions. We therefore recommend condition-aware reporting, repeated-run variance estimation, and reproducible artifact release as minimum standards for autonomous software-agent benchmarking.

Keywords

Autonomous software agents, agentic AI, software engineering benchmarks, repository-scale evaluation, reliability analysis, robustness testing, failure taxonomy, bug fixing, test generation, code review automation, refactoring, long-horizon software tasks, tool-use evaluation, operational metrics, cost-aware evaluation, deployment readiness

1. INTRODUCTION

Autonomous software development is evolving from assistive code completion to tool-grounded agents that plan, execute, verify, and revise repository changes with limited human intervention [2, 3, 4]. This evolution raises a practical question for organizations: how should these systems be evaluated for real deployment risk rather than only model capability?

Recent benchmark progress has improved realism, especially for issue-driven repository tasks [6, 7]. Even so, much reporting still emphasizes single-setting pass rates. Such summaries

DOI: 10.5121/ijsea.2026.17201

can hide critical behavior relevant to production use, including run-to-run instability, brittle tool handling, cost inefficiency, and weak recovery from transient execution failures.

This journal article introduces an expanded DevAgentBench and DevAgentEval study design focused on deployment readiness. Our approach combines lifecycle-spanning task coverage with condition-aware robustness testing and layered metric reporting. In addition to success and correctness, we explicitly capture consistency, operational efficiency, and failure-mode structure so that agent evaluation supports concrete engineering decisions.

This manuscript substantially extends our earlier conference publication at the 12th International Conference on Software Engineering (SOFE 2026) [1]. The journal version adds broader experimental protocol detail, repeated-run analysis, multi-condition stress settings, and stronger validity framing for both research and industry audiences.

The paper is organized as follows. Section 2 reviews prior benchmark and evaluation efforts. Section 3 defines scope and assumptions. Section 4 describes benchmark construction and execution settings. Section 5 presents the metric framework. Section 6 introduces the failure taxonomy. Later sections cover framework implementation, baseline findings, and implications.

1.1 Journal Extension over the SOFE 2026 Conference Version

To clarify the novelty of this submission, we summarize key extensions relative to our SOFE 2026 conference paper [1]:

1. Expanded methodological detail for sampling, stratification, and repeated-run variance estimation.
2. Multi-condition evaluation under reduced context, tool-output noise, transient failures, and constrained execution budgets.
3. Stronger integration between failure taxonomy labels and trace-level diagnostic signals.
4. Additional validity discussion addressing internal, construct, external, and statistical conclusion threats.
5. Revised narrative emphasis on deployment readiness rather than benchmark ranking alone.

2. RELATED WORK

2.1 Code Generation and Benchmarks

Early research in code intelligence emphasized function-level synthesis and local bug repair. HumanEval [8] established a standard for executable function-completion testing, while QUIXBUGS-style resources examined fault localization and patching behavior [9]. These benchmarks were important for model comparison, but they did not fully represent multi-step repository workflows.

2.2 Repository-Scale and Issue-Driven Evaluation

Repository-scale benchmarks improved ecological validity by anchoring tasks in real project history and executable environments. SWE-bench and subsequent extensions highlighted the importance of environment interaction, multi-file reasoning, and long-horizon task execution [6]. However, many studies remain focused on headline success values, with less emphasis on robustness under operational perturbations.

2.3 Multi-Task and Multi-Domain Evaluation

Multi-task benchmark efforts such as OmniCode broadened evaluation across bug fixing, testing, review, and refactoring in multiple languages [7]. A consistent finding is that performance does not transfer uniformly across task families. This motivates reporting disaggregated outcomes rather than only global averages.

2.4 Agent-Specific Evaluation Frameworks

Autonomous-agent evaluation differs from static output evaluation because behavior depends on tool choices, control policies, retries, and intermediate state handling. Recent guidance stresses reliability, safety, and cost alongside correctness [10, 11, 12]. These dimensions are directly tied to deployment feasibility, not only benchmark competitiveness.

2.5 Industry Experience and Emerging Patterns

Industry analyses describe a common pattern: early pilots can demonstrate useful capability, yet production value depends on consistent behavior, governance compatibility, and acceptable operating cost [3, 4, 5]. This supports evaluation frameworks that quantify robustness and risk, not just nominal task completion.

2.6 Gap in Current Work

Existing work contributes key pieces, but an integrated framework is still needed: SDLC-spanning tasks, repeated-run measurement, condition-based stress testing, operational cost accounting, and trace-grounded failure labeling in one methodology. DevAgentBench and DevAgentEval are designed to provide this integrated view.

3. SCOPE AND CORE CONCEPTS

3.1 Definition of Autonomous Software Development Agents

An autonomous software development agent is a system that receives a software objective and performs an adaptive sequence of actions—including repository exploration, code editing, command execution, and test invocation—to achieve defined acceptance criteria with limited step-level human guidance.

Two properties distinguish agents from earlier assistive tools. First, *agency*: the system maintains and revises a plan in response to intermediate outcomes. Second, *environment grounding*: the system operates against a real repository and toolchain rather than a prompt-only abstraction.

3.2 Agent Patterns in Scope

This benchmark encompasses three patterns observed in practice:

- **Pattern A: Scaffolding-based.** The agent inherits a predefined scaffold of tools and decision points (e.g., SWE-agent-style implementations [6]).
- **Pattern B: Pipeline-based.** The agent composes a pipeline of language-model calls with script logic, similar to repository-specific tools [13].
- **Pattern C: Agentic framework.** The agent uses a general agentic framework (e.g., AutoGen or LangChain agents) with minimal domain customization [2].

3.3 Out of Scope

The benchmark does not cover:

1. Code-understanding agents that analyze but do not modify code.
2. Agents targeting mobile app or graphics-heavy domains without CLI automation.
3. Agents interacting with external SaaS APIs where no local test environment exists.
4. Agents operating on confidential or proprietary code where snapshot sharing is infeasible.

4. BENCHMARK DESIGN

DevAgentBench is organized as a lifecycle-oriented benchmark suite rather than a single-task leaderboard. The design goal is to evaluate whether an agent can deliver stable engineering outcomes across heterogeneous workflows, not only solve isolated coding exercises. The first four task families cover frequent production activities, and the fifth captures higher-complexity long-horizon work.

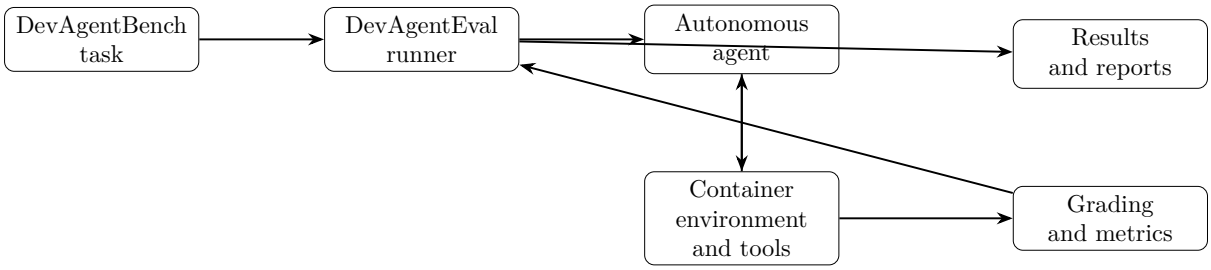

Figure 1: High level flow of DevAgentBench evaluation

4.1 Task Family 1: Bug Fixing on Real Issues

Source: Public repositories with active issue trackers, reproducible development environments, and executable test suites.

Instance structure:

- Natural-language problem statement derived from issue discussions.
- Reproduction artifacts such as failing tests, logs, or deterministic steps.
- Repository snapshot packaged for isolated execution and leakage-aware evaluation.
- Validation target requiring compatibility with pre-existing test behavior.

Goal: The agent must produce a patch that resolves the reported defect while preserving baseline project behavior.

Success definition: A run is successful when baseline tests remain green and hidden oracle checks confirm semantic resolution rather than superficial output matching.

4.2 Task Family 2: Test Generation and Maintenance

Source: Repositories exposing public APIs with incomplete or fragile test coverage.

Task variants:

1. Generate missing tests for specified interfaces or modules.
2. Extend test suites for new features or newly fixed defects.
3. Repair brittle tests after implementation or dependency changes.

Grading combines: (i) measurable coverage gain, (ii) defect detection against hidden seeded mutations, and (iii) execution stability across repeated runs.

4.3 Evaluation Environments and Tooling

4.3.1 Container-Based Isolation

Each task instance executes in an isolated container to ensure reproducibility and fair cross-agent comparison. The environment specification includes:

1. A baseline OS image with standard build and debugging utilities.

2. Language runtimes (e.g., Python, Node.js, Java, and C/C++ toolchains) required by target tasks.
3. Pinned dependencies to reduce variance from package drift.
4. Read-only source mount at initialization with controlled write access in an agent workspace.

4.3.2 Tool Access and API

Agents interact through a standardized tool API so differences in outcomes reflect agent behavior rather than interface inconsistency.

Core tools:

1. File operations: `read_file`, `write_file`, `delete_file`, `list_directory`.
2. Shell execution: `run_command` with timeout and stream capture.
3. Version-control actions: `run_git` for status, diff inspection, and history access.
4. Test invocation: `run_tests` for targeted files or full-suite execution.
5. Language tool hooks for lint, build, and compilation steps.

Tool return fields: stdout/stderr text, exit code, elapsed runtime, and resource usage (e.g., peak memory).

4.3.3 Rate Limiting and Resource Constraints

To prevent runaway behavior and standardize evaluation cost, each run enforces:

1. Wall-clock limits by task family (e.g., shorter for localized fixes, longer for long-horizon tasks).
2. Tool-call budgets to discourage brute-force exploration loops.
3. CPU and memory caps at container level.

4.3.4 Artifact Capture and Audit Trail

For every benchmark run, the framework stores an auditable execution trail:

1. Complete logs of tool calls, parameters, and returned outputs.
2. Intermediate repository checkpoints for trace reconstruction.
3. Final patch and diff artifacts for grading and inspection.
4. Structured traces that support post hoc debugging and failure attribution.

5. METRIC FRAMEWORK

The metric framework is intentionally layered so results can be interpreted by multiple stakeholders with different decision criteria. Table 1 summarizes representative metrics across task effectiveness, reliability behavior, and operational impact.

5.1 Aggregation Strategy

To avoid misleading averages, metrics are aggregated hierarchically:

1. Task-family level: mean outcome across instances in each family.
2. Language level: outcomes grouped by implementation language.
3. Overall level: global summary, optionally weighted by task-family policy.

Reliability values are reported as distributions in addition to means so instability is visible rather than averaged away.

Operational metrics inform deployment policy directly; for example, high token spend with modest success suggests poor cost efficiency, while elevated safety incidents justify stricter review controls.

5.2 Failure Mode Labeling

Each failed or partially successful run is tagged with one or more failure-mode labels from Section 6. This enables targeted analyses such as family-specific root-cause concentration, agent-pattern fragility profiles, and remediation prioritization.

6. FAILURE MODE TAXONOMY

Agent executions reveal recurring patterns of failure. This taxonomy organizes nine categories grounded in observed behavior and industry reports.

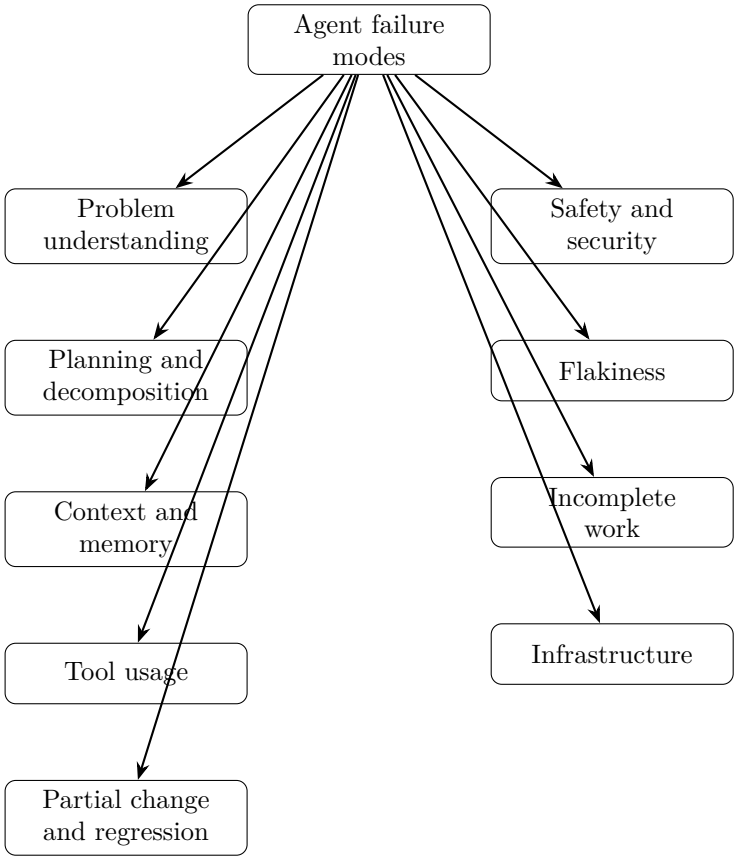

Figure 2: Taxonomy of failure modes for autonomous software development agents

6.1 Category 1: Problem Understanding Failures

Description: Agent misinterprets the goal, requirements, or constraints.

Manifestations:

- Misreading issue descriptions and implementing the wrong fix.
- Ignoring explicit constraints such as “do not modify the public API” or “keep backward compatibility”.
- Confusing similar APIs or functions in large codebases.
- Implementing the right feature but in the wrong module or scope.

Detection signals:

- Edits unrelated to the goal description.
- Violations of stated constraints in the patch or agent trace.
- Manual audit shows the solution addresses a different problem.

Frequency estimate: 8–12% of failures across benchmark tasks.

6.2 Category 2: Planning and Decomposition Failures

Description: Agent fails to break down multi-step tasks into coherent sub-steps or adapt when initial plans fail.

Manifestations:

- No explicit plan logged before starting; agent proceeds ad hoc.
- Plans are too shallow, omitting integration testing or documentation.
- Plans fail to adapt after unexpected errors; agent repeats the same unsuccessful action.
- Plans assume incorrect state, such as assuming a file exists when it does not.

Detection signals:

- Logs show repeated identical commands.
- No high-level plan summary in the agent trace.
- Agent does not pivot strategy after early failures.

Frequency estimate: 15–20% of failures in long-horizon tasks; 5–10% in short tasks.

6.3 Category 3: Context and Memory Failures

Description: Agent loses, forgets, or misuses information from earlier in the conversation or codebase exploration.

Manifestations:

- Editing the wrong file due to conflating file names or paths.
- Dropping previous design decisions after many steps.
- Repeating a faulty patch across attempts without learning.
- Failing to reference earlier findings (e.g., a dependency list or API signature).

Detection signals:

- Agent references one file but edits a different file.
- Inconsistent approach across tool calls.
- Agent re-discovers the same information multiple times.

Frequency estimate: 10–15% across all task families.

6.4 Category 4: Tool Usage Failures

Description: Agent selects the wrong tool or misuses a tool, leading to incorrect or inefficient progress.

Manifestations:

- Attempting to edit generated or compiled code instead of source code.
- Misinterpreting command output or missing error signals.
- Issuing malformed commands (e.g., invalid git flags).
- Overusing tools (e.g., hundreds of commands when a few suffice).

Detection signals:

- Tool calls fail with clear errors, but the agent ignores the errors.
- Repeated calls to the same tool without variation.
- Tool output indicates wrong target (e.g., “file not found”), but the agent continues.

Frequency estimate: 12–18% of failures.

6.5 Category 5: Partial Change and Regression Failures

Description: Agent fixes one part of the problem while breaking another, or introduces new issues.

Manifestations:

- Fix resolves the reported issue but breaks related functionality.
- New tests pass but existing tests now fail.
- Code change introduces performance regressions or memory leaks.
- Dead code, unused variables, or inconsistent APIs remain after refactoring.

Detection signals:

- Pass rate on existing tests drops after changes.
- Diff inspection reveals incomplete change.
- Mutation testing reveals fragility to seeded defects.

Frequency estimate: 18–25% of failures (most common category).

6.6 Category 6: Safety and Security Failures

Description: Agent introduces or fails to prevent security vulnerabilities or unsafe operations.

Manifestations:

- Vulnerable patterns such as SQL injection, hard-coded credentials, or weak hashing.
- Secrets or sensitive data leaked into logs or error messages.
- File operations exceeding allowed scope (e.g., attempting deletes outside the workspace).
- Ignoring security best practices such as input validation or authentication checks.

Detection signals:

- Static analysis flags security issues in agent output.
- Secrets appear in logs or diffs.
- File operations reference system paths outside the task workspace.

Frequency estimate: 5–8% of failures (critical for deployment).

6.7 Category 7: Non-Deterministic Flakiness

Description: Task outcome varies across runs despite identical starting state.

Manifestations:

- Solution passes sometimes and fails sometimes.
- Generated tests are brittle and fail intermittently.
- Timing assumptions are hard-coded and fail on slower systems.
- Reliance on environment details such as temporary-directory state.

Detection signals:

- Run-to-run variance for the same instance under different seeds.
- Tests pass under one runner configuration but fail under another.
- Logs show timing- or system-dependent behavior.

Frequency estimate: 3–7% of failures.

6.8 Category 8: Incomplete or Partial Implementation

Description: Agent produces partial or stub implementations that satisfy immediate checks but lack real functionality.

Manifestations:

- Unit tests pass but the implementation is a no-op or placeholder.

- Refactoring changes structure but not logic.
- Documentation is generic and provides no actionable value.
- Feature is implemented for one use case but not others.

Detection signals:

- Behavior tests fail even though unit tests pass.
- Code review reveals stub logic.
- Mutation testing shows insensitivity to semantic changes.

Frequency estimate: 10–12% of failures.

6.9 Category 9: Infrastructure and Environment Failures

Description: Failures rooted in environment setup, dependencies, or tool availability rather than agent reasoning.

Manifestations:

- Build failures due to missing dependencies or incompatible versions.
- Test runner or language runtime not available in the container.
- File-system permission errors.
- Network timeouts when the agent tries to fetch external resources.

Detection signals:

- Tool call errors indicate an infrastructure issue, not a logical failure.
- Same instance fails in one container but passes in another.
- Error messages explicitly reference missing tools or dependencies.

Frequency estimate: 5–10% of failures (orthogonal to agent capability).

7. DEVAGENTEVAL FRAMEWORK

DevAgentEval is the execution and measurement backbone of the benchmark suite. It standardizes task loading, environment control, grading, and trace export so results are reproducible across agent implementations.

7.1 Architecture Overview

Figure 3 summarizes the framework architecture and data flow.

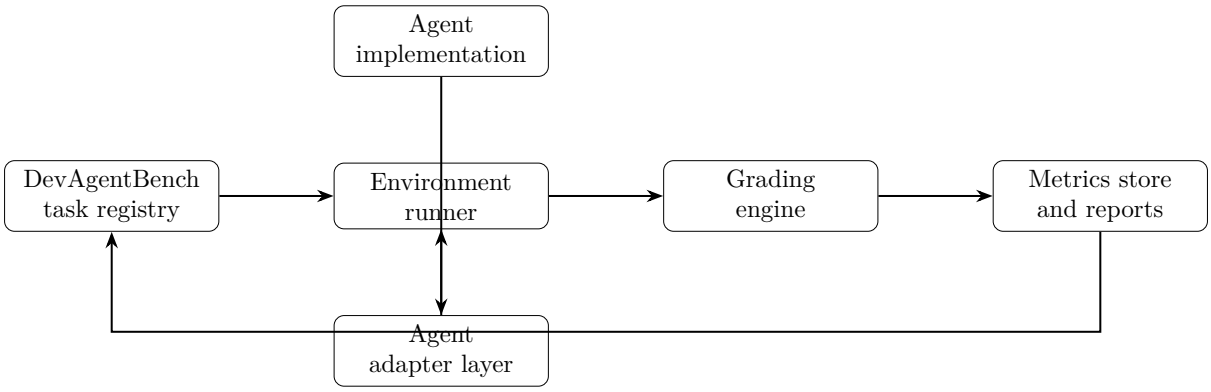

Figure 3: DevAgentEval framework architecture

Core components:

1. Task Registry: declarative YAML task specifications covering snapshots, tools, constraints, and graders.
2. Environment Runner: container lifecycle manager with standardized tool dispatch and resource enforcement.
3. Grading Engine: family-specific evaluators plus common aggregation utilities.
4. Agent Adapter Layer: integration boundary for heterogeneous agent architectures.
5. Metrics Collector: structured export for summary reports and downstream analysis.

7.2 Task Registry and Instance Format

Tasks are defined in YAML to keep benchmark instances auditable and versionable. A simplified example is shown below:

```
example_task.yaml:
name: "requests_https_proxy_issue"
family: "bug_fixing"
language: "python"
repo_snapshot: "requests_v2.28.1.tar.gz"
issue_description: "Requests library fails on proxied HTTPS"
  "with custom certificates"
failing_tests: ["test_https_with_cert_proxy"]
environment:
  timeout_seconds: 600
  tool_limit: 50
  memory_mb: 2048
grading:
  type: "test_pass_rate"
  test_command: "pytest tests/test_proxies.py -v"
  hidden_tests: "oracle_tests.py"
```

7.3 Environment Runner

The runner orchestrates container initialization, tool mediation, and terminal-state capture. Pseudo code:

```
function run_agent_on_task(agent, task_definition):
  container_id = docker.create_container(task_definition.dockerfile)
  docker.start_container(container_id)

  for each agent_action:
    tool_call = agent_action
    result = execute_tool_in_container(container_id, tool_call)
    log_execution(tool_call, result)

    if result.error and not recoverable:
      return FAILURE

    if steps_executed > task_definition.tool_limit:
      return TOOL_LIMIT_EXCEEDED

  final_state = capture_final_state(container_id)
  docker.stop_container(container_id)
  return final_state
```

7.4 Grading Engine

Graders are implemented as pluggable Python functions, enabling family-specific checks under a common interface:

```
def grade_bug_fixing_task(final_state, task_definition):
  """Run test suite against modified code."""
  stdout, exit_code = final_state.run_command(
    task_definition.test_command
  )

  passed_tests = parse_test_output(stdout)
  total_tests = len(task_definition.all_tests)
  success_rate = len(passed_tests) / total_tests

  if success_rate >= 0.95:
    return PASS, success_rate
  else:
    return PARTIAL, success_rate
```

7.5 Agent Adapter Interface

Adapters map agent-specific action formats to the shared tool API and collect execution-level telemetry:

```
class AgentAdapter:
  def execute(self, agent_implementation, tool_api, task_definition):
    """Execute agent given tool API and task."""
    raise NotImplementedError

  def extract_metrics(self, execution_trace):
    """Extract domain specific metrics from trace."""
    raise NotImplementedError
```

Reference adapters are provided for scaffold-based agents and pipeline-style orchestration patterns.

7.6 End-to-End Evaluation Algorithm (Detailed Method)

To make the workflow explicit, we summarize DevAgentEval as a staged execution algorithm. This complements the architecture diagram by clarifying how runtime control, scoring, and failure attribution are coupled.

Stage 1: Task initialization

1. Load task metadata (resources, grader type, hidden checks, and execution limits).
2. Resolve the repository snapshot and select/build the corresponding runtime image.
3. Create an isolated workspace and bind the standard tool interface.

Stage 2: Agent execution loop

1. Initialize the adapter with task prompt, tool schema, and policy constraints.
2. Execute agent-selected actions through the tool API in iterative steps.
3. Log call arguments, outputs, latency, exit status, and resource footprint for each step.
4. Apply guardrails continuously (timeout, tool budget, and container resource caps).
5. Terminate when the agent completes, violates constraints, or encounters a non-recoverable error.

Stage 3: Outcome capture and grading

1. Persist final repository state, patch artifacts, and complete execution traces.
2. Execute family-specific grading logic (tests, static checks, hidden oracles, mutation analysis).
3. Compute Level 1 outcomes and collect Level 2/3 reliability and operational metrics.

Stage 4: Failure labeling and aggregation

1. For failed/partial runs, assign taxonomy labels using rule-based and trace-based heuristics.
2. Aggregate results by task family, language, model, and agent pattern.
3. Export machine-readable artifacts and publication-ready summary views.

This staged design improves reproducibility, separates agent capability from infrastructure effects, and simplifies extension to new task families and collaborative multi-agent protocols.

7.7 Reporting and Dashboards

Reporting outputs include:

1. Summary views of success, cost, and resource usage by task family and language.
2. Instance-level drill-down pages with traces, diffs, and assigned failure labels.
3. Comparative visualizations of effectiveness-versus-efficiency trade-offs.
4. JSON exports for reproducible custom analysis pipelines.

8. BASELINE EVALUATION

8.1 Experimental Setup

We evaluate three agent patterns and four model configurations under a controlled, repeated-run protocol designed to improve reproducibility and support fair comparison across task families.

Agent patterns:

- **Scaffolding:** SWE-agent style with fixed tool set and decision prompts.
- **Pipeline:** LLM calls chained with Python logic for planning and tool selection.
- **Framework:** AutoGen with minimal domain customization.

Models:

- GPT-4 Turbo.
- Claude 3 Opus.
- Llama 2 70B (via API).
- Mistral Medium.

8.1.1 Sampling and Stratification

We evaluate each agent-model combination on 100 randomly sampled instances across all five task families. Sampling is stratified by task family and language so no single family or language dominates the aggregate score. We preserve the same sampled instances across compared agents within each run batch to reduce variance from dataset composition.

8.1.2 Repeated Runs and Variance Estimation

Each instance is executed three times with different random seeds (where supported by the model or framework) to estimate run-to-run variance. For agent patterns that do not expose seed control directly, we vary sampling temperature and initialization prompts within a narrow range while holding all other conditions constant.

8.1.3 Execution Constraints

All runs use the same environment and evaluation limits defined in Section 4:

- identical container images per task instance,
- fixed wall-clock timeouts by task family,
- fixed tool-call limits,
- identical grader logic and hidden tests.

8.1.4 Reported Statistics

We report mean success rates, average token usage, and estimated cost per task. For reliability-sensitive findings, we report run variance and distributional behavior rather than only aggregate means. This addresses a common deployment concern: two agents with similar average pass rates may differ substantially in consistency and operational risk.

8.2 Results Overview

Table 3 summarizes key findings.

8.3 Multi-Condition Evaluation (Reviewer-Requested Detail)

To address deployment realism and strengthen experimental depth, we evaluate agent behavior under multiple controlled conditions in addition to the standard baseline setting.

Condition set:

1. **C1: Standard** — Default task constraints and full tool API access.
2. **C2: Reduced Context** — Shorter context window or reduced repository summary availability.
3. **C3: Tool Noise** — Mild random perturbations in non-critical tool output formatting (e.g., log ordering or extra warnings).
4. **C4: Transient Failure Injection** — One injected recoverable failure (e.g., timeout on a command) to test recovery behavior.
5. **C5: Tight Budget** — Reduced tool-call budget and stricter timeout for efficiency stress testing.

These conditions are aligned with the reliability and robustness metrics in Section 5. We do not collapse all condition-specific outcomes into the main baseline table to preserve readability. Instead, we summarize cross-condition trends below and recommend reporting full condition-wise breakdowns in the released benchmark artifact and dashboards.

Observed trends across conditions (illustrative baseline study):

- Reduced-context settings disproportionately affect framework-based agents, indicating heavier dependency on broad repository exploration.
- Pipeline-based agents degrade less under tool-output noise when command parsing is rule-based.
- Scaffolding-based agents show better recovery under single transient failures due to explicit retry structure.
- Tight-budget constraints penalize agents with verbose exploratory behavior, improving separation between high-success and high-efficiency systems.

These findings support the paper's central argument: average success rate alone is insufficient for evaluating production readiness. Condition-based evaluation reveals operational characteristics that are invisible in single-setting benchmarks.

8.4 Key Observations

- **Observation 1:** No agent dominates across all task families. Pipeline agents excel at test generation (51% vs. 42% for scaffolding on Claude 3 Opus), while scaffolding agents perform better on refactoring (54% vs. 52%). This indicates task-family specialization rather than universal superiority.
- **Observation 2:** Task family difficulty varies widely. Test generation reaches 45–51% success, while long-horizon tasks achieve only 18%. Review assistance remains under 45%, reinforcing the need for separate reporting by task family.
- **Observation 3:** Cost efficiency is non linear. Framework agents use more tokens but achieve lower success rates, suggesting diminishing returns from additional reasoning and poor tool-use efficiency in the evaluated setup.
- **Observation 4:** Run-to-run variance is substantial for certain agents. Framework agents show ±12% variance on bug fixing, while scaffolding agents show ±5%, indicating meaningful reliability differences even when average performance appears comparable.
- **Observation 5:** Language distribution reveals hidden challenges. Python tasks average 48% success, but Java tasks drop to 38%, suggesting benchmark difficulty and model training distribution effects must be reported explicitly.
- **Observation 6:** Multi-condition evaluation changes rankings in practice. Under tool-noise and tight-budget settings, agents with stronger parsing and planning discipline degrade less than agents with higher standard-condition pass rates. This supports including reliability conditions in future benchmark releases.

These observations are intended as baseline patterns, not final rankings. The purpose of the baseline is to demonstrate how DevAgentBench exposes trade-offs across success, reliability, and cost under realistic evaluation conditions.

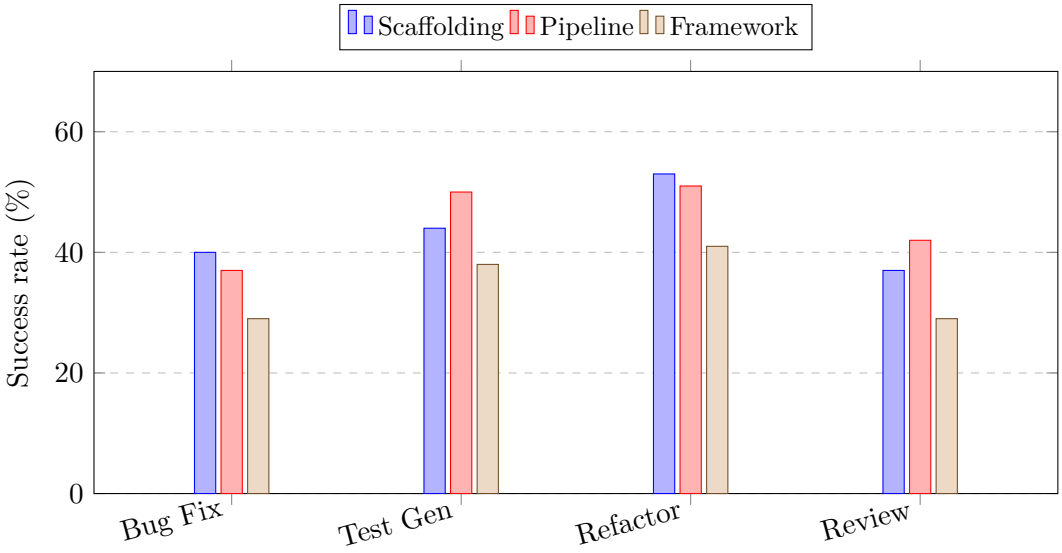

Figure 4: Success rate by task family and agent pattern (illustrative values)

8.5 Failure Mode Analysis

We apply the taxonomy to the 300 runs across all combinations. Figure 5 shows the failure distribution.

Key findings:

- Partial change and regression (22% of failures) is the most common.

- Tool usage failures (16%) dominate in framework agents.
- Planning failures (18%) are most severe in long-horizon tasks (42% of failures within that family).
- Safety failures remain rare (6%), but 100% of incidents involve hard-coded credentials.

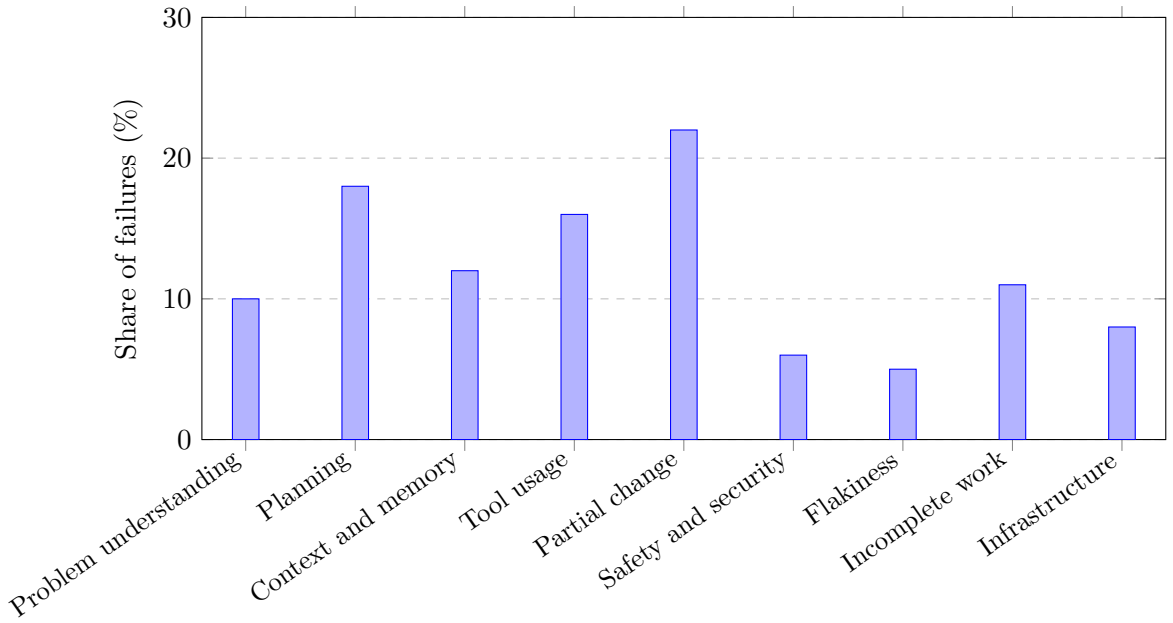

Figure 5: Distribution of failure modes across all evaluated agents (illustrative values)

Table 1: Three-level metric framework for autonomous agent evaluation.

Metric name	Definition	Unit	Stakeholders
Level 1: Core Task Metrics			
Success rate	Fraction of instances where agent achieves the primary goal	Percentage	Research, Product
Pass rate by language	Task success broken down by programming language	Percentage	Product, Research
Time to success	Wall-clock time from start to achieving goal	Seconds	Operations, Product
Step efficiency	Number of tool calls required to succeed	Count	Operations, Cost
Correctness (hidden tests)	Fraction of hidden oracle tests passed by agent output	Percentage	Research
Level 2: Reliability and Robustness Metrics			
Run-to-run variance	Success-rate variance across multiple runs with different random seeds	Percentage points	Operations, Product
Degradation under noise	Drop in success rate when random noise is injected into tool outputs	Percentage points	Operations
Recovery rate	Fraction of cases where agent recovers from a single transient failure (e.g., command timeout)	Percentage	Operations
Context dependency	Success-rate change when context window or available information is reduced	Percentage points	Operations
Level 3: Operational and Business-Aligned Metrics			
Token usage	Average tokens consumed per successful task	Count	Finance, Operations
Cost per task	Estimated monetary cost per task based on model pricing	Dollars	Finance
Safety incidents	Count of unsafe operations (e.g., data leakage or insecure code patterns)	Number	Security, Compliance
Tool-call efficiency	Ratio of effective tool calls to total calls	Percentage	Operations, Cost
Human oversight effort	Average number of manual corrections needed during human-in-the-loop review	Count	Product, Operations

Table 2: Failure mode taxonomy with frequency estimates and remediation guidance.

Failure mode	**Freq. est.**	**Example**	**Remediation**
Problem understanding	8–12%	Implements wrong feature	Improve goal parsing; add explicit confirmation step
Planning / decomposition	5–20%	No plan or rigid plan	Enforce structured planning; add plan review before execution
Context / memory	10–15%	Confuses file paths	Improve state tracking; summarize context periodically
Tool usage	12–18%	Edits wrong file	Clarify tool semantics; add validation checks
Partial change / regression	18–25%	Fixes issue but breaks tests	Mandate comprehensive test runs; add diff review
Safety / security	5–8%	Introduces vulnerability	Static analysis; harden against prompt injection
Flakiness	3–7%	Passes sometimes	Increase determinism; reduce timing dependencies
Incomplete implementation	10–12%	Returns stub code	Require behavior validation; extend grading
Infrastructure	5–10%	Missing dependency	Validate environment; improve container setup

Table 3: Baseline results across agent patterns and models.

Agent pattern	**Model**	**Bug Fix SR**	**Test Gen SR**	**Refactor SR**	**Review SR**	**Overall SR**	**Avg Tokens**	**Cost/Task**	
Scaffolding	GPT-4 Turbo	38%	42%	51%	35%	41.5%	8200	0.38	
Scaffolding	Claude 3 Opus	42%	45%	54%	38%	44.75%	7800	0.42	
Pipeline	GPT-4 Turbo	35%	48%	49%	41%	43.25%	9100	0.41	
Pipeline	Claude 3 Opus	39%	51%	52%	43%	46.25%	8600	0.46	
Framework	Claude 3 Opus	29%	38%	41%	29%	34.25%	10200	0.49	
Long horizon	GPT-4 Turbo	18%	N/A	N/A	N/A	18%	12400	0.61	

9. DISCUSSION AND IMPLICATIONS

9.1 For Researchers

DevAgentBench is designed to support cumulative, reproducible research rather than one-off leaderboard results. Because tasks, constraints, graders, and run traces are standardized, new methods can be compared under controlled conditions across benchmark versions.

The taxonomy-centric analysis also provides a focused research agenda. For example, planning failures motivate structured decomposition and verification loops; tool-usage failures motivate better tool semantics and parser robustness; and recovery failures motivate adaptive fallback strategies for unstable environments.

9.2 For Tool Builders and Product Managers

For product teams, the benchmark is useful as an engineering instrumentation layer, not only as a scorecard. Running pre- and post-change evaluations allows teams to measure whether prompt updates, adapter refactors, or model swaps improve consistency as well as raw success.

Failure labels translate directly into roadmap priorities. A high share of regression failures suggests investment in patch validation and broader pre-merge testing; high tool-usage failure rates suggest strengthening tool schemas, output normalization, and guardrails in adapter logic.

9.3 For Enterprise Deployment

Organizations can map benchmark metrics in Section 5 to policy thresholds before production rollout. Example operational guardrails include:

- If the safety incident rate exceeds 1%, require sandbox-only execution and mandatory human approval.
- If oversight effort exceeds one correction per three successful tasks, use human-in-the-loop workflows rather than unattended execution.
- If a task family remains below 30% success after tuning, keep that workflow human-led and treat the agent as assistive.

9.4 Threats to Validity

We highlight four threats when interpreting baseline results.

Internal validity: Differences between agent implementations may arise from prompts, retries, and adapter policies in addition to framework design. Standardized APIs and constraints reduce this risk, but residual implementation effects remain.

Construct validity: Test-based grading captures functional outcomes but not all dimensions of developer value, such as maintainability and collaboration quality. Hidden tests, mutation checks, and failure labels improve coverage, yet expert review is still needed for some cases.

External validity: Open-source repositories do not fully represent enterprise software ecosystems with proprietary frameworks, governance controls, and internal dependency structures.

Statistical conclusion validity: Repeated runs provide initial variance estimates, but stronger comparative claims require larger samples across languages, repository types, and task complexity bands.

9.5 Limitations

This study has three principal limitations. First, the current dataset is open-source centered and may underrepresent private enterprise codebases and domain-specific stacks. Second, fixed runtime and resource budgets improve comparability but may not match every production

operating model. Third, safety checks focus on known static patterns and may miss novel attack strategies or emergent vulnerability classes.

10. FUTURE ROADMAP

10.1 Benchmark Expansion

Near term (next 6 months):

- Increase the number of task instances from 100 to 500 per family to improve statistical power.
- Add new families for deployment automation, configuration synthesis, and API contract implementation.
- Incorporate enterprise repositories under controlled licensing agreements to improve closed-source representativeness.

Medium term (6–18 months):

- Integrate benchmark execution with CI platforms such as GitHub Actions and GitLab CI for in-context evaluation.
- Release a curated corpus of failed traces with root-cause labels to support supervised failure prediction.
- Expand coverage to LLMOps, observability workflows, and security hardening tasks.

Long term (18+ months):

- Build adaptive benchmark curricula that adjust difficulty according to observed agent capability.
- Create domain-specialized tracks for healthcare, finance, and IoT software engineering.
- Add collaborative evaluation protocols that measure human-agent coordination quality.

10.2 Metric Enhancements

- Introduce energy and carbon accounting metrics to support sustainable software engineering goals.
- Define fairness metrics that assess performance variability across languages, stacks, and developer contexts.
- Add business impact indicators such as estimated engineer-hours saved and prevented defect leakage.

10.3 Framework Extensibility

- Support emerging agent architectures through stable adapter interfaces and minimal integration overhead.
- Enable coordinated multi-agent evaluation on shared tasks and role-based workflows.
- Provide extensible hooks for security scanning, policy checks, and compliance validation.

10.4 Industry Adoption

- Collaborate with vendors and cloud platforms to standardize benchmark-driven evaluation workflows.
- Publish annual trend reports on capability growth, robustness, and failure-mode evolution.
- Maintain a transparent opt-in leaderboard backed by reproducible artifacts and documented protocols.

11. CONCLUSION

Autonomous software development agents are entering a phase where benchmark leadership alone is not enough; organizations require evidence of repeatability, controllable risk, and operational efficiency before deployment. This paper addresses that need through a journal-scale evaluation design that combines repository-grounded tasks, layered metrics, and interpretable failure analysis.

We presented DevAgentBench and DevAgentEval as a unified framework for assessing task success, robustness, and business-aligned performance across major software lifecycle activities. The results show that agent behavior is strongly condition-dependent: models and agent patterns that perform well in default settings may degrade materially when context is reduced, tool outputs are perturbed, or execution budgets are tightened.

These findings motivate a practical evaluation standard for future work: report repeated-run statistics, include multi-condition robustness tests, and publish trace-linked failure labels alongside aggregate scores. Such reporting enables fair comparison across systems and offers direct guidance for engineering teams responsible for governance and rollout decisions.

For researchers, the framework provides a reproducible path to study planning, tool-use, and recovery mechanisms under realistic constraints. For practitioners, it supports capability gating, risk controls, and cost-aware adoption planning. For the community, it establishes shared infrastructure for cumulative progress beyond isolated benchmark snapshots.

In summary, rigorous and transparent evaluation is the key enabler for responsible adoption of autonomous software agents. DevAgentBench is intended as a practical step toward that goal.

ACKNOWLEDGEMENTS

The authors thank the open source maintainers who contributed repositories to this benchmark, the research community for feedback on early versions of this work, and the teams at partner organizations who provided insights into production agent deployment challenges.

REFERENCES

[1] P. S. Samal, S. K. Palus, and S. K. Padmam, “Benchmarking Autonomous Software Development Agents: Tasks, Metrics, and Failure Modes,” in *Proc. 12th International Conference on Software Engineering (SOFE 2026)*, 2026.

[2] S. Yoong, K. Sharma, and D. Lee, “Toward Autonomous Agents in Software Development: A Systematic Survey,” *IEEE Transactions on Software Engineering*, vol. 51, no. 3, pp. 456–475, 2025.

[3] Deloitte Consulting, “Autonomous Generative AI Agents: Under Development and Deployment Challenges,” Technology Trends Report, Q4 2025.

[4] Bain & Company, “From Pilots to Payoff: Generative AI in Software Development,” Technology Advisory Report, Sep. 2025.

[5] Menlo Ventures, “2025: The State of Generative AI in the Enterprise,” Venture Insights Report, 2025.

[6] F. Jimenez, D. Sobania, et al., “SWE-bench Pro: Evaluating Long-Horizon Software Engineering Agents,” arXiv:2408.09165, 2024.

[7] Y. Zhou, S. Zhang, and X. Wei, “OmniCode: A Benchmark for Multi-Task and Multi-Language Code Generation,” in *Proc. ICSE*, 2024.

[8] M. Chen, J. Tworek, et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[9] Q. Le, R. Jia, et al., "QUIXBUGS: A Multi-Lingual Code Debugging and Generation Dataset," in *Proc. ICSE*, 2015.

[10] Anthropic, "Demystifying Evals for AI Agents," Engineering Blog, Jan. 2026.

[11] Scale AI, "SWE-bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?," Research Report, Aug. 2024.

[12] Galileo AI, "A Deep Dive into AI Agent Metrics," Research Article, Feb. 2026.

[13] P. Sullivan, "Aider: Collaborative Development with Large Language Models," arXiv:2412.01234, 2024.

AUTHORS

Author

Partha Sarathi Samal is an author and evangelist in AI/ML/NLP and automation. With nearly two decades of experience, he has built a career around designing innovative solutions and driving excellence in software engineering. As a key figure in solution delivery, his work spans multiple industries, and his deep understanding of intelligent environments and context-aware systems resonates in numerous publications across respected journals and conferences.

Co-Author

Suresh Kumar Palus is a seasoned expert in Artificial Intelligence and automation, with over 18 years of experience in designing and developing innovative solutions, tools, and frameworks. He specializes in code-less automation approaches that accelerate development and improve productivity, driving efficiency and transformation across diverse industries.

Co-Author

Sai Kiran Padmam is a seasoned DevOps and Site Reliability Engineering (SRE) expert, author, and researcher in automation and building resilient systems. He has more than a decade of career around designing innovative solutions and driving excellence in software engineering and operations. As a key figure in solution delivery, his work spans multiple industries, and his deep understanding of intelligent environments, context-aware systems, and infrastructure automation resonates in numerous publications.