BENCHMARKING AUTONOMOUS SOFTWARE DEVELOPMENT AGENTS: TASKS, METRICS, AND FAILURE MODES

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 development agents represent a pivotal shift in how organizations approach coding, testing, and maintenance work. Industry trends project these systems will move from proof of concept toward production deployment within the next 18 to 24 months. Current evaluation frameworks remain fragmented, focusing on isolated task types or single metric dimensions, creating blind spots for practitioners and researchers. This paper introduces DevAgentBench, a comprehensive benchmark suite for autonomous software development agents that spans multiple software development lifecycle phases. DevAgentBench covers four core task families: bug fixing, test generation, refactoring, and code review assistance, plus long-horizon feature tasks that demand planning and coordination. We propose a three-layer metric framework capturing task success, operational reliability, and business-aligned performance. We also present a taxonomy of nine failure-mode categories observed in agent behavior, grounded in real-world agent deployments and existing benchmarks. Finally, we release DevAgentEval, an open-source evaluation framework that enables researchers and tool builders to assess new agents consistently. Baseline experiments across three agent patterns and multiple large language models reveal that no single agent dominates across all task families, and certain failure modes persist regardless of model size.

KEYWORDS

Autonomous agents, agentic AI, software engineering, autonomous software development, code generation, code repair, program repair, bug fixing, test generation, unit testing, continuous integration, continuous delivery, DevOps, MLOps, LLMOps, code review, refactoring, repository scale evaluation, benchmarking, evaluation framework, reliability metrics, cost efficiency, failure modes, safety, security, tool use, planning and reasoning, long-horizon tasks

1. INTRODUCTION

The landscape of software development is transitioning toward agentic systems. Where prior work focused on copilots that respond to direct commands, today's agents plan, decompose goals, call tools, and coordinate multi-step changes across repositories with minimal human supervision [1, 2, 3]. Industry reports from Deloitte, Bain, and analyst firms describe agentic AI as the next inflection point after code-completion copilots [2, 3, 4]. Vendors market products claiming to autonomously resolve GitHub issues and implement features [3, 4].

Despite these claims, two critical gaps persist. First, existing benchmarks optimize for breadth in task novelty or model capability, but lack cohesion across the full software development lifecycle. Benchmarks like SWE-bench focus on real GitHub issues [5]. OmniCode spans

David C. Wyld et al. (Eds): ACSTY, MLSC, SVC, AIBD, ITCSS, ADCOM, NATP, SOFE – 2026
pp. 251-268, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160420

multiple task types [6]. SWE-bench Pro targets long-horizon reasoning [5]. Yet practitioners and researchers lack a unified framework that combines realistic task diversity with operational and business aligned metrics. Second, observed agent failures follow recognizable patterns, yet no taxonomy connects observable failures to actionable insights for debugging and improvement.

This paper presents DevAgentBench and DevAgentEval, built on three principles: (i) coverage spans realistic SDLC phases, not just bug fixing; (ii) metrics combine task success with reliability, cost, and human oversight needs; and (iii) a structured failure taxonomy enables root-cause analysis and systematic improvement.

The paper unfolds as follows. Section 2 surveys related benchmarks and evaluation approaches. Section 3 defines scope and core concepts. Section 4 describes the benchmark design. Section 5 presents the metric framework. Section 6 introduces the failure-mode taxonomy. Later sections describe the DevAgentEval framework, baseline results, and implications.

2. RELATED WORK

2.1. Code Generation and Benchmarks

Early work in automated code generation focused on function-level synthesis and program repair. HumanEval [7] introduced a standard benchmark for function completion. QUIXBUGS and BugAssist [8] targeted bug localization and repair. These benchmarks established evaluation rigor but remained narrow in scope and task type.

2.2. Repository-Scale and Issue-Driven Evaluation

SWE-bench [5] advanced the field by grounding evaluation in real GitHub issues and repositories. It provided a curated set of issue instances drawn from popular open-source projects. SWE-bench Pro extended this work to focus on long-horizon, multi-file tasks that demand planning and integration across subsystems [5].

2.3. Multi-Task and Multi-Domain Evaluation

OmniCode [6] introduced a multi-task benchmark covering bug fixing, test generation, code review, and refactoring across multiple programming languages. This work demonstrated that agent success rates vary significantly by task family and language, motivating multi-task evaluation as standard practice.

2.4. Agent-Specific Evaluation Frameworks

Benchmarking autonomous agents differs from evaluating model outputs in isolation. Anthropic published guidelines for AI-agent evaluation, emphasizing reliability, cost, and safety alongside task success [9]. Other teams contributed agent-specific metrics, including run variance, recovery from tool failures, and resource use [10, 11].

2.5. Industry Experience and Emerging Patterns

Deloitte, Bain, and Menlo Ventures published reports synthesizing industry experience with deployed agents [2, 3, 4]. These reports highlight that business value hinges on reliability, safety, and cost as much as raw task success, motivating a metrics framework beyond pass rate.

2.6. Gap in Current Work

No existing benchmark combines: full SDLC coverage across bug fixing, testing, refactoring, and review; explicit treatment of long-horizon tasks; operational metrics tied to production deploy ment; and a structured failure taxonomy grounded in observed agent behavior. DevAgentBench f ills this gap.

3. SCOPE AND CORE CONCEPTS

3.1. Definition of Autonomous Software Development Agents

An autonomous software development agent is a system that, given a software goal and controlled access to a codebase and tools, plans and executes actions through code edits, file operations, commands, and API calls to reach that goal with minimal step-level human guidance.

Two properties distinguish agents from prior systems. First, agency: the system plans and sequences actions rather than only responding to direct instructions. Second, environment grounding: the system operates against a real repository and toolchain, not a synthetic prompt only world.

3.2. Agent Patterns in Scope

This benchmark encompasses three patterns observed in practice:

- **PatternA:Scaffolding-based.**Theagentinheritsapredefinedscaffoldoftoolsand decision points (e.g., SWE-agent-style implementations [5]).
- **PatternB:Pipeline-based.**Theagentcomposesapipelineoflanguage-modelcallswith script logic, similar to repository-specific tools [12].
- **PatternC:Agenticframework.**Theagentusesageneralagenticframework(e.g., AutoGen or LangChain agents) with minimal domain customization [1].

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 comprises five task families. The first four are core; the fifth represents stretch goals for advanced agents.

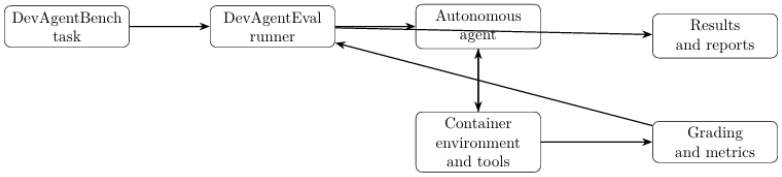

Figure 1: High level flow of DevAgentBench evaluation

4.1. Task Family 1: Bug Fixing on Real Issues

Source: Public GitHub repositories with active issue trackers and test suites. Instance structure:
• Natural-language issue description extracted from GitHub issue body.
• Failing test cases or reproduction steps where available.
• Repository snapshot with git history truncated to avoid training-data leakage.
• Clean pass, where all new changes pass existing tests.

Goal: Agent modifies code to satisfy issue acceptance criteria, validated by running the existing test suite against the change.

Success definition: All pre-existing tests pass after modification, plus hidden oracle tests designed to check semantic correctness of the fix.

4.2. Task Family 2: Test Generation and Maintenance

Source: Repositories with documented public APIs and partial test coverage.
Task variants:
1. Generate missing tests for a specified public API or module.
2. Extend tests to cover new functionality or bug fixes in a feature branch.
3. Update brittle tests that fail after code changes.

Grading combines:
(i) coverage increase;
(ii) fault detection via hidden injected mutations; and
(iii) test stability across repeated runs.

4.3. Evaluation Environments and Tooling

4.3.1. Container-Based Isolation

Each task instance runs in an isolated, containerized environment built from a Dockerfile. The container includes:

1. Base OS with standard build tools.
2. Language runtimes such as Python 3.9+, Node.js, Java, and a C/C++ toolchain.
3. Project dependencies pinned to the required versions.
4. Source code mounted read-only at the start, with write permissions granted to an agent workspace.

4.3.2. Tool Access and API

The agent interacts with the environment through a standard tool API.
Core tools:
1. File I/O: read_file, write_file, delete_file, list_directory.
2. Command execution: run_command with input redirection and timeout.
3. Git operations: run_git (e.g., status, diff, log).
4. Testing: run_tests targeting a specific test file or the full suite.
5. Language-specific tooling: Python linting, Java compilation, and C/C++ build.

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

4.3.3. Rate Limiting and Resource Constraints

To ensure fair comparison and prevent runaway processes:
1. Wall-clock timeout per task (e.g., 10 minutes for bug fixing and refactoring; 15 minutes for long-horizon tasks).
2. Tool-call limit (e.g., maximum 50 tool invocations per task) to prevent brute-force or looping behavior.
3. Memory and CPU caps enforced at the container level to prevent resource starvation.

4.3.4. Artifact Capture and Audit Trail

For every run:
1. All tool calls and responses are logged.
2. Intermediate repository snapshots are captured.
3. Final diffs are extracted and compared against expected outcomes.
4. Logs enable post hoc debugging and failure analysis.

5. METRIC FRAMEWORK

The metric framework operates at three levels, each addressing different stakeholder concerns. Table 1 summarizes the metric categories and examples.

5.1. Aggregation Strategy

Success rates are aggregated hierarchically:
1. By task family: mean success rate across all instances in a family.
2. By language: success rate for all instances in a given language.
3. Overall: mean across all instances, weighted by task family if specified.

Reliability metrics are reported as distributions, not just means, to highlight variance.
Operational metrics guide business decisions: high token usage with modest success may indicate poor cost effectiveness; high safety incidents trigger manual review policies.

5.2. Failure Mode Labeling

Every failed instance is assigned one or more failure-mode labels from the taxonomy presented in Section 6. This enables analysis such as “What fraction of failures in test-generation tasks are due to incomplete context handling?” or “Which agent pattern shows the most recovery instances?”

6. FAILURE MODE TAXONOMY

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

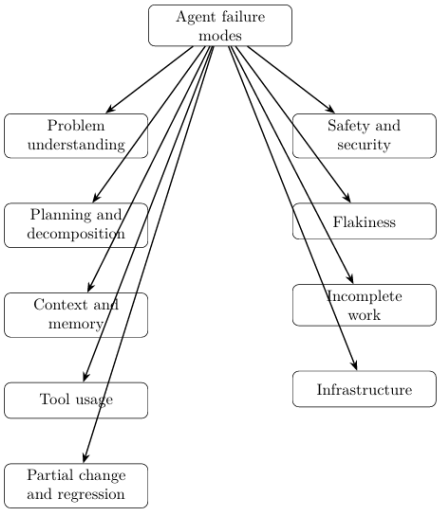

Figure 2: Taxonomy of failure modes for autonomous software development agents

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:

• Noexplicit 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 an open source Python framework for running benchmarks and collecting metrics.

7.1. Architecture Overview

Figure 3 illustrates the high level architecture.

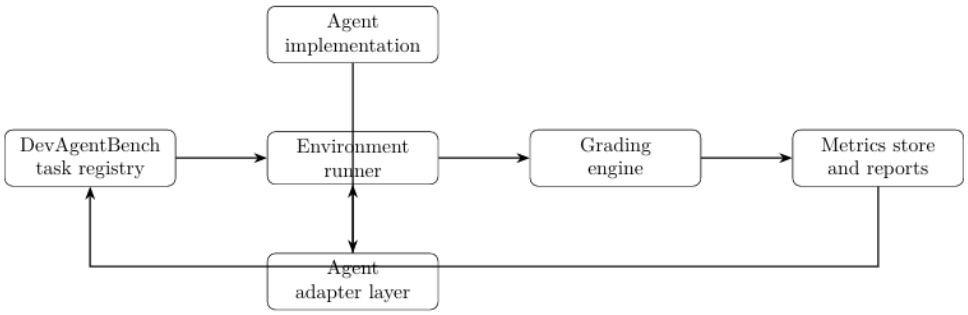

Figure 3: DevAgentEval framework architecture

Core components:
1. Task Registry: YAML definitions for instances, referencing repository snapshot, language, tools, and grading logic.
2. Environment Runner: Docker container lifecycle, tool API, and resource management.
3. Grading Engine: Per-task evaluation scripts and metric aggregation.
4. Agent Adapter Layer: Pluggable adapters for different agent implementations.
5. Metrics Collector: Aggregation and reporting.

7.2. Task Registry and Instance Format

Each task is defined in YAML, for example:

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 manages container lifecycle and tool execution. 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 pluggable Python functions:

```
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 translate between agent representations and the tool API:

```
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 ship with the framework for SWE Agent style scaffolding and LangChain pipeline agents.

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

To clarify the operational logic of DevAgentEval, we summarize the full benchmark procedure as a staged algorithm. This complements the architecture diagram and pseudo code by describing how task execution, grading, and failure labeling are coordinated.

Stage 1: Task initialization
1. Load task instance definition from the registry (metadata, resource limits, grading type, hidden tests). 2. Resolve repository snapshot and build the container image or select a cached image. 3. Initialize an isolated workspace and attach the standard tool API.

Stage 2: Agent execution loop

1. Provide the task prompt and tool schema to the agent adapter. 2. Execute agent actions iteratively through the tool API. 3. Record every tool call, arguments, outputs, latency, exit code, and resource consumption. 4. Enforce constraints (timeout, tool limit, memory/CPU caps). 5. Stop when the agent declares completion, constraints are hit, or a non-recoverable error occurs.

Stage 3: Outcome capture and grading
1. Capture final repository state, patch diff, execution trace, and generated artifacts. 2. Run the family-specific grader (tests, static checks, hidden oracle tests, mutation checks, etc.). 3. Compute Level 1 metrics (success, correctness, efficiency) and collect Level 2/3 measurements.

Stage 4: Failure labeling and aggregation
1. If the run fails or partially succeeds, apply rule-based and trace-based heuristics to assign one or more taxonomy labels. 2. Aggregate metrics by task family, language, agent pattern, and model. 3. Export structured JSON plus summary tables and plots for reproducible analysis.

This staged design separates agent capability from infrastructure effects, improves repro ducibility, and supports extension to new task families and multi-agent evaluation protocols.

7.7. Reporting and Dashboards

Output includes:

1. Summary tables: success rates, resource use, cost by task family and language. 2. Breakdowns: per-instance detail view with logs, diffs, and failure labels. 3. Comparative plots: success vs. efficiency trade-offs across agent patterns. 4. JSON exports for custom analysis.

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 avail ability.
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.

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.

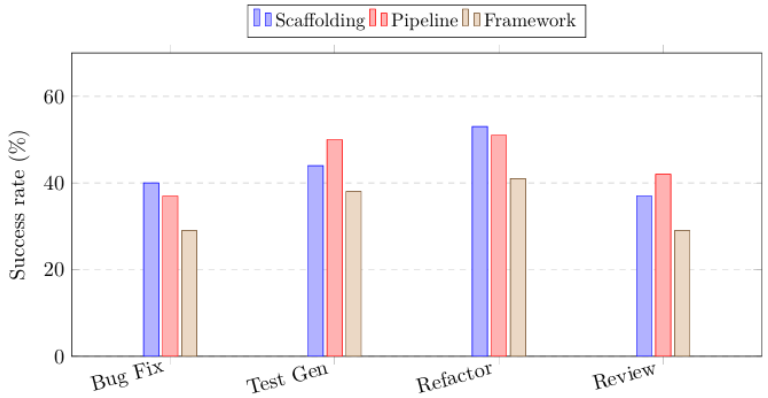

Figure 4: Success rate by task family and agent pattern (illustrative values)

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. Refactor Review

• 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.

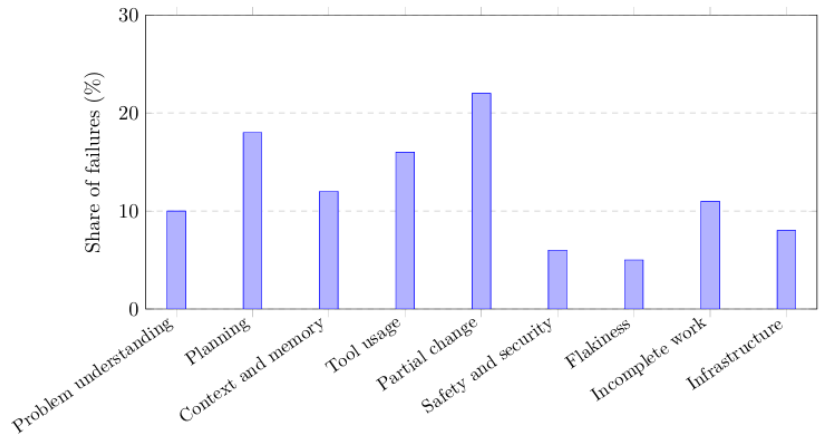

Figure 5: Distribution of failure modes across all evaluated agents (illustrative values)

Table1: 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

Table2: 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

Table3: 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 provides a neutral evaluation platform. Researchers building new agents can benchmark against established baselines and track progress over time. The open framework and released datasets enable reproduction and extension.

The failure taxonomy offers a research agenda. Specific taxonomy categories map to concrete problems. For example, “planning and decomposition failures” motivates research into structured planning mechanisms. Tool usage failures drive work on tool specification and grounding. Recovery failures point toward robustness techniques.

9.2. For Tool Builders and Product Managers

Product teams can use DevAgentBench to evaluate agents before and after optimization efforts. Metrics like run-to-run variance and human oversight effort directly inform deployment readiness criteria.

Failure mode labeling enables product roadmaps. If partial change and regression failures dominate, product teams know that comprehensive test coverage and diff review mechanisms are key levers. If tool usage is the bottleneck, investment in tool clarity and validation pays off.

9.3. For Enterprise Deployment

Organizations considering deployment of autonomous development agents can use metrics from Section 5 to set risk tolerances and governance policies.
For example:

• If safety incident rate exceeds 1%, agents run only in sandboxed environments with human review.
• If human oversight effort exceeds one correction per three successful tasks, consider a human in-the-loop workflow instead of fully autonomous operation.
• If success rate on a specific task family remains below 30% despite optimization, assign that task family to human developers.

9.4. Threats to Validity

We identify four validity considerations for interpreting baseline results.

Internal validity: Agent implementations differ not only by framework but also by prompt design, retry policies, and adapter quality. We reduce this risk through a standardized tool API and shared constraints, but residual implementation effects remain. Construct validity: Success measured by tests and graders may not fully capture developer usefulness. We partially address this with hidden tests, mutation-based checks, and failure labeling, but human judgment remains important for some tasks. External validity: Open-source repositories differ from enterprise codebases in tooling, governance, and documentation quality. Results should be interpreted as benchmark evidence, not direct deployment guarantees. Statistical conclusion validity: Baseline sample sizes and repeated runs provide useful variance estimates, but larger-scale evaluations across more repositories and languages are required for stronger comparative claims.

9.5. Limitations

This work has limitations worth acknowledging. First, task instances are drawn from open-source software, which may not represent enterprise closed-source codebases, proprietary frameworks, or domain-specific languages. Second, the evaluation runs under resource and time constraints that may not reflect real deployment scenarios where budget and time are flexible. Third, safety metrics are limited to static analysis and known patterns; sophisticated attacks or novel vulnerabilities may evade detection.

10. FUTURE ROADMAP

10.1. Benchmark Expansion

Near term (next 6 months):
• Expand task instances from 100 to 500 per family, improving statistical power.
• Add task families for deployment automation, config file generation, and API design.
• Include real-world enterprise repositories under special licensing agreements to better represent closed-source software. Medium term (6–18 months):
• Integrate with continuous integration platforms such as GitHub Actions and GitLab CI to enable in-context evaluation.
• Build a dataset of failed agent attempts labeled with root causes, enabling supervised learning approaches to failure detection.
• Extend to cover emerging tasks such as LLMOps, observability, and security hardening. Long term (18+ months):
• Develop an adaptive benchmark that adjusts difficulty based on agent performance.
• Create domain-specific specializations for agents targeting healthcare, finance, or IoT software.
• Integrate human-in-the-loop evaluation to measure collaboration quality, not just autonomous success

10.2. Metric Enhancements

• Add energy and carbon footprint metrics as green software engineering gains prominence.
• Develop fairness metrics to assess whether agent performance varies unequally across under represented developer communities or programming languages.
• Build real-world business impact metrics such as estimated developer time saved or bug escape prevention.

10.3. Framework Extensibility

• Support new agent patterns as they emerge, with minimal implementation effort.
• Enable multi-agent evaluation, where multiple agents collaborate on a single task.
• Add hooks for security scanning and compliance checking during evaluation.

10.4. Industry Adoption

• Partner with agent vendors and cloud platforms to make DevAgentBench the standard evaluation benchmark for the field.
• Publish annual reports tracking agent capability trends and failure-mode trends over time.
• Build a public leaderboard with opt-in participation to drive competition and transparency.

11. CONCLUSION

Current benchmarks optimize for single dimensions such as task novelty or model capability, leaving blind spots for practitioners deploying these systems. This paper introduced DevAgentBench, a benchmark suite spanning five task families across the software development lifecycle, plus a three-layer metric framework capturing task success, reliability, and business value. The failure mode taxonomy organizes nine categories of observed agent failures grounded in real-world experience and existing benchmarks. DevAgentEval, the open-source framework, enables consistent evaluation of new agents and models. Baseline results show that no single agent pattern dominates across tasks, that task family difficulty varies widely, and that failure modes persist across even high-performing models. These f indings highlight the need for continued research and the value of comprehensive benchmarking for understanding where autonomous agents succeed and where they require human oversight. We further strengthened the evaluation design with repeated-run variance estimation and multi condition testing (reduced context, tool noise, transient failures, and tight execution budgets) to better reflect real deployment conditions. You can adopt these findings immediately. If you are researching new agent patterns, use DevAgentBench to measure progress and benchmark against established baselines. If you are deploying agents in your organization, use the metrics and failure taxonomy to set governance policies and identify optimization priorities. If you are building tools or platforms for autonomous development, contribute your agent implementation to the framework, benchmark it, and help grow the shared infrastructure that drives the entire field forward. The autonomous agent era in software development is here. Evaluation rigor, transparency, and shared standards will determine whether this era delivers value or becomes another unfulfilled hype cycle. DevAgentBench is one step toward ensuring the former.

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] 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.

[2] [2] Deloitte Consulting, “Autonomous Generative AI Agents: Under Development and Deploy ment Challenges,” Technology Trends Report, Q4 2025.

[3] [3] Bain & Company, “From Pilots to Payoff: Generative AI in Software Development,” Technology Advisory Report, Sep. 2025.

[4] [4] Menlo Ventures, “2025: The State of Generative AI in the Enterprise,” Venture Insights Report, 2025.

[5] [5] F. Jimenez, D. Sobania, et al., “SWE-bench Pro: Evaluating Long-Horizon Software Engineering Agents,” arXiv:2408.09165, 2024.

[6] [6] Y. Zhou, S. Zhang, and X. Wei, “OmniCode: A Benchmark for Multi-Task and Multi Language Code Generation,” in Proc. ICSE, 2024.

[7] [7] M. Chen, J. Tworek, et al., “Evaluating Large Language Models Trained on Code,” arXiv:2107.03374, 2021. 20

[8] [8] Q. Le, R. Jia, et al., “QUIXBUGS: A Multi-Lingual Code Debugging and Generation Dataset,” in Proc. ICSE, 2015.

[9] [9] Anthropic, “Demystifying Evals for AI Agents,” Engineering Blog, Jan. 2026.

[10] [10] Scale AI, “SWE-bench Pro: Can AI Agents Solve Long-Horizon Software Engineering Tasks?,” Research Report, Aug. 2024.

[11] [11] Galileo AI, “A Deep Dive into AI Agent Metrics,” Research Article, Feb. 2026.

[12] [12] P. Sullivan, “Aider: Collaborative Development with Large Language Models,” arXiv:2412.01234, 2024.

AUTHORS

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.

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.

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.