

Autonomous AI Agents for Software Development: From Code Generation to Automated Testing

An Agentic Framework for Plan-Code-Test-Repair Workflows

Nikalje Aryan Gautam

Independent Research / Student Research Paper
Email: aryangnikalje@gmail.com | Mobile: 7420899266
Guide: Mr. Suryawanshi A. M.

Abstract

Large language models (LLMs) have moved from code completion toward systems that can read a repository, plan changes, run commands, execute tests and revise their own work. This paper studies Autonomous AI Agents for Software Development and proposes a modular, role-based framework that covers the path from requirement understanding and code generation to automated test generation, debugging and review. The framework separates planning, context retrieval, coding, tool execution, testing, debugging and review into explicit roles connected by a bounded feedback loop: if tests fail or review checks are not satisfied, the planner revises the approach, otherwise the system delivers a patch with a test report and an audit trace. The paper synthesizes prior work on code-generating language models, reasoning-and-acting agents, self-debugging, repository-level issue resolution, multi-agent software teams and LLM-based unit test generation. It also presents an evaluation protocol covering functional correctness, test quality, efficiency and cost, safety, and component ablations. Because no implementation or benchmark run is reported here, the comparison charts are explicitly conceptual and the evaluation plan is a proposal rather than a claim of measured improvement. The contribution is an auditable design blueprint for building and evaluating autonomous software-development agents.

Index Terms — Autonomous agents, large language models, AI software engineering, code generation, automated testing, program repair, tool use, multi-agent systems, SWE-bench, evaluation.

I. INTRODUCTION

Software development involves far more than writing individual functions. Engineers read unfamiliar code, locate the files affected by a requirement, edit several modules, run builds and tests, interpret failures and iterate until the change is acceptable. Early neural code tools helped with only a small part of this process: given a prompt, they produced a completion. Models trained on code showed that functional correctness could be measured by running unit tests on generated programs [1], and later benchmarks showed that solving real repository issues is considerably harder than solving isolated programming problems [3].

An autonomous software agent wraps a language model in a loop with tools. Instead of answering once, it can search a codebase, edit files, execute a shell command, read the error output and decide what to do next. The ReAct pattern demonstrated that interleaving reasoning with actions lets a model gather information and correct itself [5]. Reflection and self-debugging showed that execution feedback can improve generated code across turns [6], [7]. SWE-agent studied how the design of the agent-computer interface affects performance on repository tasks [4], while MetaGPT and ChatDev explored teams of role-specialized agents that imitate software-process roles [8], [9].

Autonomy is not the same as unlimited freedom. A code-writing agent that can run commands can also delete files, leak secrets, install untrusted packages or submit incorrect changes with high confidence. A dependable design therefore needs sandboxed execution, explicit tool permissions, bounded iteration, human approval for consequential actions and logs that allow every step to be audited.

This paper does not claim a new benchmark result. Its objective is to define a reproducible architecture and evaluation protocol. The main contributions are: (1) a role-based architecture for autonomous software-development agents spanning code generation and automated testing; (2) a plan-code-test-repair control loop with explicit stopping conditions; (3) an evaluation framework measuring correctness, test quality, efficiency, safety and component contribution; and (4) a discussion of security, limitations and deployment considerations.

II. BACKGROUND AND LITERATURE REVIEW

A. Code Generation with Language Models

Chen et al. introduced Codex and the HumanEval benchmark, evaluating generated Python functions by executing unit tests and reporting pass@k, the probability that at least one of k samples passes [1]. Austin et al. studied program synthesis with large models on the MBPP and MathQA-Python datasets [2]. Code Llama released open code-specialized models [14], and EvalPlus showed that augmenting benchmark tests exposes incorrect solutions that the original tests miss [12]. These works establish an important principle that carries into agentic systems: generated code should be judged by execution, not by how plausible it looks.

B. Reasoning, Acting and Tool Use

ReAct interleaves reasoning traces with actions in an external environment [5]. Toolformer studied how a model can learn when to call external APIs [15]. In software agents the same pattern applies to shell commands, file editors, search, version control and test runners. Tool use must be constrained: arguments should be validated, permissions should be explicit and irreversible actions should require human confirmation.

C. Reflection and Self-Debugging

Reflexion stores verbal feedback from earlier attempts and uses it in later ones [6]. Self-Debugging teaches a model to inspect execution results and explain its own code in order to find errors, reporting gains on several code generation benchmarks [7]. These approaches motivate a debugger role that consumes test output and stack traces rather than guessing a fix from the original prompt alone.

D. Repository-Level Issue Resolution

SWE-bench collects real GitHub issues and pull requests from popular Python repositories and asks a system to produce a patch that makes the project's tests pass [3]. SWE-agent introduced a customized agent-computer interface to help a model navigate, edit and test code [4]. AutoCodeRover combines program-structure-aware code search with fault localization [11], and Agentless argued that a simpler pipeline of localization, repair and validation can be competitive with complex agent loops [13]. OpenHands provides an open platform in which agents write code, use a command line and browse the web inside a sandbox [10]. Together these works show that design choices such as context retrieval, interface design and validation strategy matter as much as the underlying model.

E. Multi-Agent Software Teams

ChatDev organizes agents into roles such as programmer, tester and reviewer that communicate through a software development process [9]. MetaGPT encodes standard operating procedures and structured intermediate artifacts to reduce inconsistent communication among role agents [8]. Role separation makes responsibilities and failure points explicit, although more agents also mean more calls, more tokens and more coordination errors.

F. Automated Unit Test Generation

Schäfer et al. evaluated LLM-driven unit test generation for JavaScript and reported that generated tests can reach meaningful coverage, while also noting that correctness and flakiness of generated tests must be checked [16]. Test generation matters in an agent for two reasons: tests can reproduce a reported bug before a fix is attempted, and tests provide the feedback signal for the repair loop. A weak or wrong test can, however, mislead the whole loop.

G. Research Gap and Design Motivation

The literature shows complementary capabilities — generation, planning, tool use, reflection, localization, role separation and test generation — but a practitioner still has to decide how to combine them, how to limit repeated attempts and how to evaluate the trade-off between quality and cost. Highly autonomous workflows increase adaptability but also increase latency, token consumption, tool errors, security exposure and debugging difficulty. The proposed framework addresses this engineering gap through explicit roles, test-based verification, bounded retries and log-oriented evaluation.

III. PROBLEM STATEMENT AND OBJECTIVES

The problem addressed is the brittleness of single-shot code generation when a task requires repository context, multiple edits, execution feedback or verification. A system that returns code without running it risks syntax errors, hallucinated APIs, regressions and untested edge cases. Conversely, an agent that retries without a stopping rule may consume large budgets and still fail, or may overwrite correct code while chasing a faulty test.

Objectives:

- Decompose a requirement or issue into steps while preserving the original intent.
- Retrieve only the repository context that is relevant to the change.
- Generate or edit code through controlled tools inside an isolated sandbox.
- Generate, execute and triage tests, distinguishing code defects from test defects.
- Use a bounded repair loop that stops on success, on low expected benefit, or when a budget is reached.
- Deliver a patch with a test report, rationale and audit trace, with human review for consequential actions.
- Evaluate correctness, test quality, efficiency, cost and safety against baselines.

IV. PROPOSED AUTONOMOUS SOFTWARE-DEVELOPMENT AGENT

The system is modular. Components may be separate agents or logically separated prompts and functions inside one orchestration service; the design does not require every role to be a different model. Role separation exists mainly to make responsibilities, logs and failure handling explicit.

A. Context Sources and Ingestion

Context may include the code repository and its history, documentation and specifications, an issue tracker, the existing test suite and continuous-integration logs, and approved package or web documentation. The ingestion step builds a repository map (files, symbols, dependencies) and indexes code and documents, following retrieval-augmented generation [18], with both lexical search, which is useful for identifiers, and embedding search, which is useful for natural-language descriptions. Each chunk keeps a stable identifier, file path and commit reference so that every claim in the final report can be traced to a source. Access controls apply at retrieval time: the agent must not read files outside its authorized scope.

B. Planner Agent

The planner interprets the request, classifies the task (new feature, bug fix, refactor, test addition) and produces a short, revisable plan with sub-tasks and acceptance criteria. Simple, well-localized tasks take a short path; multi-file or ambiguous tasks activate more steps or trigger a clarification question to the developer.

C. Context Retriever

The retriever returns the files, functions and tests most relevant to the current sub-task, together with metadata. It prefers structure-aware lookups (symbol definitions, call sites, imports) and falls back to hybrid text and embedding search. Keeping the context small and relevant is important because a large context is not automatically a useful one.

D. Coder Agent

The coder produces edits as minimal, reviewable diffs rather than rewriting whole files. It follows project conventions found in the retrieved context, and every edit is applied through the tool layer so that it can be logged and reverted.

E. Tool Layer and Sandbox

All actions run inside an isolated environment with allow-listed tools: file read and edit, code search, shell commands, linter, type checker, package installation from approved registries, git operations and the test runner. Each call is schema-validated and logged with arguments, exit status, duration and output summary. Output from tools, web pages and repository files is treated as data, not as instructions; text inside a file that asks the agent to ignore its rules must not change its behavior.

F. Tester Agent

The tester has two jobs. First, for a bug report it writes a failing reproduction test before any fix is attempted. Second, after a change it generates additional unit and edge-case tests, executes the existing suite and collects pass/fail results and coverage. Fig. 4 shows this sub-pipeline.

G. Debugger Agent

When tests fail, the debugger reads the stack trace, failing assertion and relevant code, forms a hypothesis about the root cause and proposes a targeted repair. It also performs *triage*: a failure may come from a defect in the code or from a faulty generated test. Without this step an agent may repeatedly “fix” correct code to satisfy an incorrect test.

H. Reviewer Agent

The reviewer checks the final diff against the plan, runs static analysis, looks for unrelated changes, hard-coded secrets, unsafe patterns and violated policies, and confirms that the stated acceptance criteria are covered by tests. Review by a model is a verification stage, not a guarantee of correctness; it shares blind spots with the coder and must be evaluated independently.

I. Bounded Re-Planning and Stopping Rules

If tests fail or review checks are not met, the planner may revise the plan, request different context, or ask the developer a question. The loop is bounded by a maximum number of repair rounds, a time budget, a token budget and a tool policy. It stops when the acceptance gate passes, when further attempts are unlikely to help (for example the same failure repeats), or when a budget is exhausted. In the last case the agent returns its best partial work with a clear limitation statement rather than claiming success.

J. Output, Memory and Feedback

The final output contains a code patch, a test report, a pull-request description with rationale and an audit trace listing actions, cost and unresolved risks. Short-term memory holds the current plan and observations; long-term memory may hold project conventions and past decisions, subject to privacy and retention rules. Developer approvals and rejections are stored as feedback for evaluation and policy tuning.

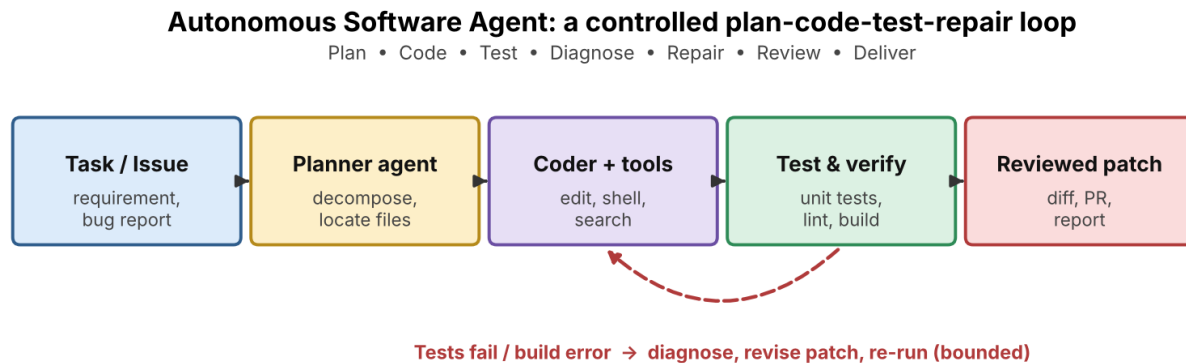


Fig. 1. Conceptual illustration of the proposed plan-code-test-repair loop. The feedback path is bounded by retry and resource limits.

V. ARCHITECTURE AND WORKFLOW

The architecture follows a data path and a control path. The data path carries repository files, retrieved context, diffs, test logs and the final artifacts. The control path carries the plan, tool choices, acceptance decisions, retry decisions and stopping conditions. Keeping them separate helps debugging and auditing: a failed task can be traced to context retrieval, planning, code generation, test generation, execution or review.

Proposed Autonomous Software-Development Agent: Architecture

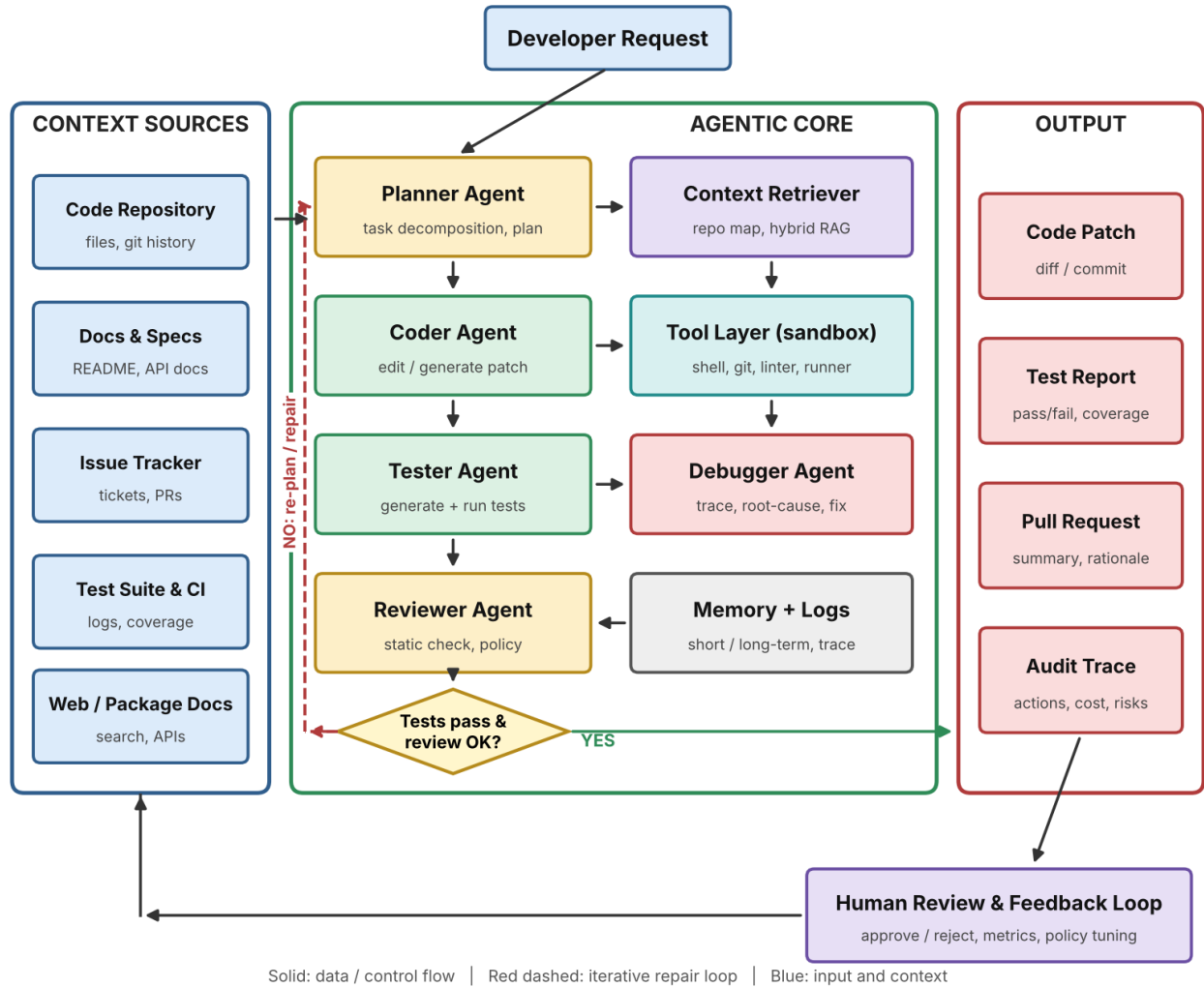


Fig. 2. Proposed architecture of the autonomous software-development agent, from context sources through the agentic core to outputs and the human feedback loop.

In an implementation each transition should have a typed input/output contract. For example, the retriever returns file path, line range, snippet, score and commit reference; the tester returns a structured result containing test identifiers, status, error message and coverage; and the acceptance gate returns structured reasons so that the planner can choose a targeted next action instead of repeating the same attempt.

Code Assistant (Single-Shot) vs. Autonomous Agent (Iterative)

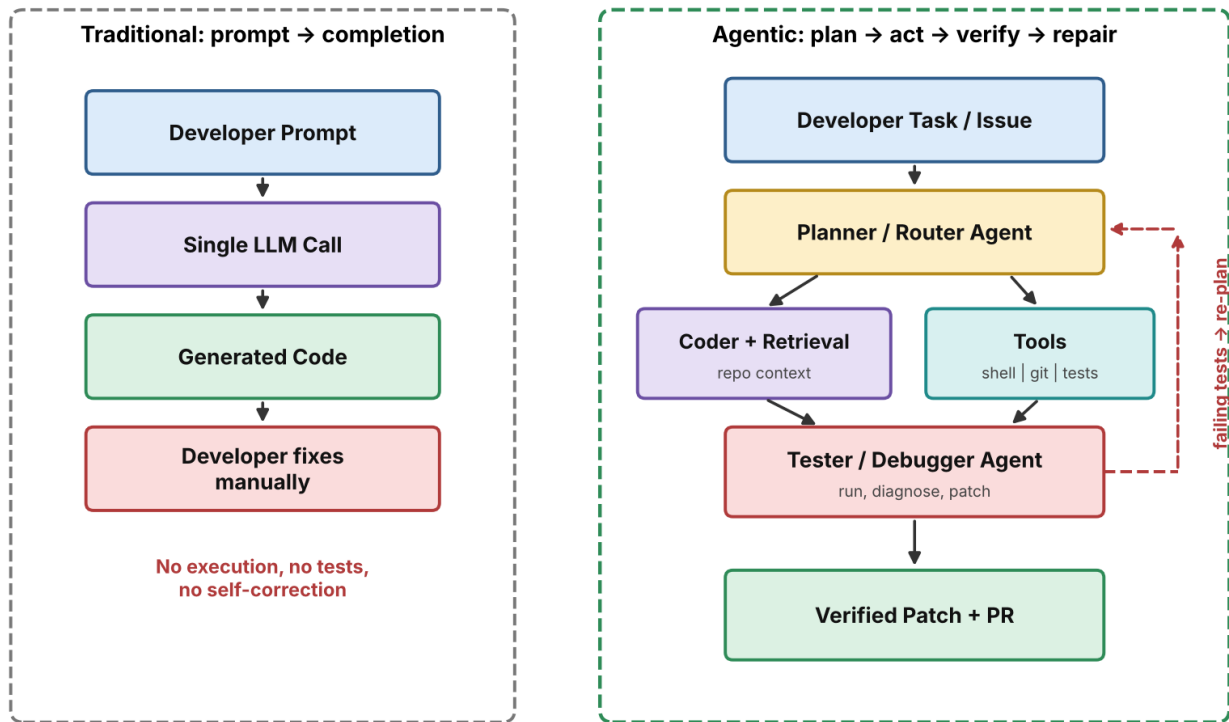


Fig. 3. Comparison of a single-shot code assistant and the iterative, goal-driven autonomous agent.

The comparison emphasizes the main conceptual distinction: a traditional assistant generates once and leaves verification to the developer, whereas an autonomous agent can plan, call tools, run tests, diagnose failures and revise. This adaptability is not free; it increases the number of components and the number of failure modes, which is why the design includes budgets, logs and explicit stopping criteria.

Automated Testing Sub-Pipeline Inside the Agent

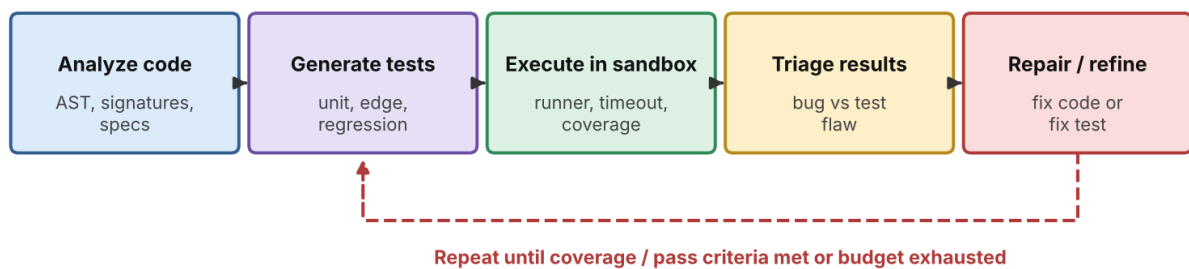


Fig. 4. Automated testing sub-pipeline: analyze, generate, execute, triage and repair.

VI. ALGORITHM AND CONTROL LOGIC

The following high-level procedure defines the control loop. It is model-agnostic and can be implemented with an orchestration framework, a custom state machine or a workflow engine.

Step	Description
Input	Task t (requirement or issue), repository snapshot R , tool registry T , retry limit N , time/token budget B .
Plan	Classify the task; create sub-tasks and acceptance criteria; select a first action.

Step	Description
Retrieve	Collect relevant files, symbols and tests with provenance metadata.
Code	Produce a minimal diff through the tool layer inside the sandbox.
Test	For bugs, add a failing reproduction test first; run existing and new tests; record results and coverage.
Branch	If all gates pass, proceed to review. If tests fail and budget remains, triage (code vs test defect), diagnose and repair. Otherwise stop.
Review	Check the diff for scope, security, policy and coverage of the acceptance criteria.
Finalize	Return patch, test report, rationale and audit trace; if gates are unmet, return the best partial work with a limitation statement.

A core invariant is that the agent never reports a task as complete unless its stated acceptance gate has passed, and never changes tests merely to make a failure disappear without recording the reason. In high-stakes applications the final patch should be reviewed by a qualified engineer, and actions that change external state (merging, deploying, deleting, publishing) should require explicit authorization.

VII. PROPOSED EVALUATION FRAMEWORK

Evaluation should compare the agent with at least three baselines: (i) the same language model without tools; (ii) single-shot generation with retrieved repository context; and (iii) a tool-using agent without the tester, debugger or reviewer roles. A fair comparison holds the model, repository snapshots, prompts, budgets and output format constant where possible, and repeats runs when model behavior is stochastic.

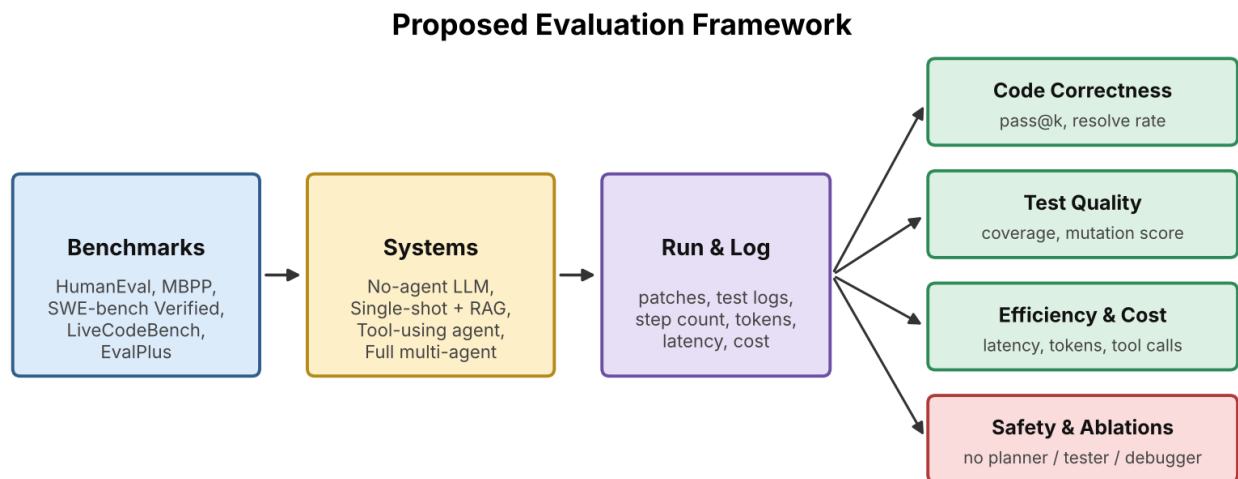


Fig. 5. Proposed evaluation framework showing benchmarks, systems, run logs and evaluation dimensions.

A. Benchmark and Test-Set Design

Function-level benchmarks such as HumanEval and MBPP [1], [2], their strengthened variants in EvalPlus [12] and contamination-aware sets such as LiveCodeBench [17] can measure code generation. Repository-level behavior can be measured with SWE-bench and its human-validated subsets [3]. A project-specific test set should additionally include bug fixes, feature additions, refactors, tasks with ambiguous requirements, tasks whose requirement is underspecified and tasks that should be refused or escalated. Each item needs a reference outcome or rubric, a difficulty label and a held-out split to avoid tuning on test items.

B. Code Correctness

Function-level tasks use pass@k [1]. Repository-level tasks use the resolve rate: the fraction of tasks for which the produced patch makes the specified failing tests pass without breaking previously passing tests [3]. Human review should additionally rate readability, minimality of the diff and adherence to project conventions. LLM-as-judge scores can assist at scale but must not be the sole measure because the judge may share biases with the generator. No values for these metrics are reported here because no experiment has been conducted.

C. Test Quality

Generated tests should be assessed by statement and branch coverage, mutation score (the fraction of injected faults detected), validity (does the test pass on the correct reference solution), flakiness rate over repeated runs, and the rate at which a reproduction test fails before the fix and passes after it.

D. Efficiency, Cost and Reliability

Efficiency measures include end-to-end latency, number of steps, tool calls, input and output tokens and estimated cost per resolved task. Reliability measures include tool-failure rate, retry rate, the percentage of tasks that hit the budget limit and the quality of abstention when a task cannot be completed. Results should be reported as averages and distributions, not only as a single best run.

E. Safety and Security

Safety checks include whether the agent stays inside its permitted paths, whether it resists prompt-injection text placed in repository files, whether it avoids leaking secrets, whether it requests approval for consequential actions and whether it avoids introducing known-vulnerable dependencies or insecure patterns.

F. Ablation Study

Ablations isolate each component: remove the planner, remove context retrieval, disable the tester, disable the debugger, remove the reviewer, restrict the tool layer, or set the retry limit to zero. A component is considered useful only if its quality gain justifies its latency and cost. A full experiment should report the same task set for all ablations and document model versions, prompts, tool configuration, repository snapshots and the date of execution.

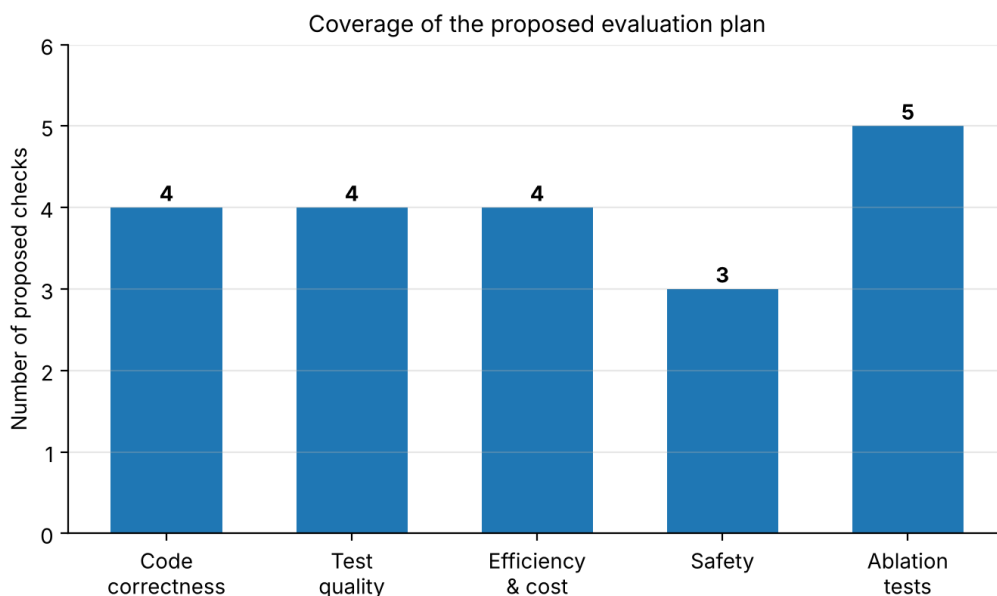


Fig. 6. Number of proposed checks in each evaluation dimension. This chart counts items in the evaluation plan; it is not a performance result.

VIII. DESIGN-LEVEL COMPARISON

The following comparison summarizes expected design characteristics, not benchmark findings. Scores are illustrative judgments on a 1–5 scale intended to make trade-offs visible; they must not be interpreted as measured system performance and should be replaced with empirical results after implementation and evaluation.

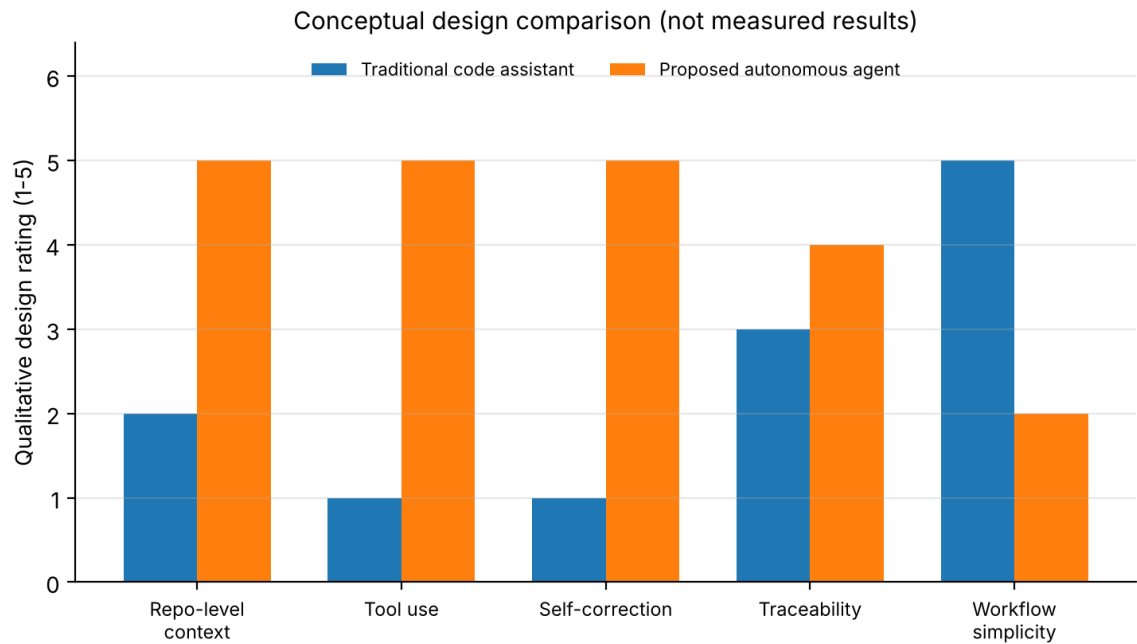


Fig. 7. Conceptual comparison of design capabilities. Ratings are illustrative and are not derived from experimental measurements.

Dimension	Traditional code assistant	Proposed autonomous agent
Control flow	Prompt in, completion out	Planner selects actions and can re-plan
Context	Current file or pasted snippet	Repository-level retrieval of relevant files and tests
Tool use	Usually none or manually wired	Sandboxed shell, git, linter, test runner
Verification	Left to the developer	Tests, triage, review gate before delivery
Latency / cost	Low and predictable	Potentially higher due to additional calls
Auditability	Depends on the host tool	Designed for step-level logs and provenance
Failure handling	Incorrect code returned as-is	Can repair, qualify the result or abstain within limits
Security exposure	Mostly text output risks	Larger: executes commands, so needs sandbox and permissions

IX. SECURITY, PRIVACY AND GOVERNANCE

An agent that can execute code introduces risks beyond ordinary text generation. Repository files, issue text, web pages and tool output may contain prompt-injection text intended to manipulate the agent, so all of them must be treated as untrusted data. The orchestration layer should use an isolated sandbox without access to production credentials, allow-listed tools, least-privilege tokens, network egress controls, input and output validation and explicit authorization for actions such as merging, deploying or deleting.

Code and logs may contain proprietary or personal information. Sensitive data should be minimized before indexing, retention and deletion policies should be defined, and logs should keep enough detail for debugging and audits without storing unnecessary secrets. Dependencies suggested by an agent should be verified against trusted registries, because a model can propose packages that do not exist or are malicious. Licensing of generated code and attribution requirements should also be reviewed by the adopting organization.

X. APPLICATIONS

- **Issue-to-pull-request assistants** that read a ticket, reproduce the bug, propose a fix and attach test evidence.
- **Automated test generation** for legacy code with low coverage, with human review of the generated tests.
- **Code migration and refactoring** (framework upgrades, API deprecations) validated by the existing test suite.
- **Documentation and code review support** that checks diffs against project conventions.
- **Education**: tutoring systems that explain failures and suggest fixes while showing the test evidence.
- **Continuous-integration repair**: diagnosing failed builds and proposing a targeted patch for human approval.

The appropriate level of autonomy depends on code criticality, test-suite quality, the cost of a wrong change and privacy constraints.

XI. LIMITATIONS

This paper proposes an architecture and evaluation protocol; it does not present an implemented system, user study, benchmark scores, statistical significance tests or a production deployment. Therefore no claim is made that the framework outperforms any baseline in measured accuracy, cost or latency. The comparison chart is conceptual and the coverage chart counts proposed checks only. Further work must implement the design, publish its configuration and dataset protocol, conduct reproducible experiments and report both improvements and regressions.

Practical limitations also apply. More calls increase cost and latency; planners can decompose a task incorrectly; generated tests can encode a wrong specification; test suites may be too weak to detect regressions, so passing tests do not prove correctness; benchmark results may not transfer to proprietary codebases; benchmark data may overlap with model training data; and retries can reinforce an initial mistaken assumption. The architecture reduces some failure modes through explicit checks but cannot eliminate defects or guarantee correct software.

XII. CONCLUSION AND FUTURE SCOPE

This paper presented a modular framework for autonomous AI agents in software development that extends single-shot code generation with planning, context retrieval, sandboxed tool use, automated test generation, debugging, review and bounded re-planning. Its central design principle is to make development adaptive while retaining verification, traceability and control. Test-based acceptance gates and a stopping policy help distinguish a verified change from code that merely looks plausible, and structured logs provide the basis for auditing and systematic evaluation.

Future work should implement the architecture and evaluate it on public and project-specific benchmarks; compare single-agent and multi-agent configurations; study test-generation quality with mutation analysis; measure the contribution of each role through ablation; analyze cost-quality trade-offs across retry budgets; and investigate prompt-injection resistance, secure sandboxing, calibrated confidence, long-horizon tasks and human-in-the-loop review. Empirical conclusions should be added only after controlled runs with documented methods and reproducible results.

REFERENCES

- [1] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
- [2] J. Austin et al., "Program Synthesis with Large Language Models," arXiv:2108.07732, 2021.
- [3] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" International Conference on Learning Representations (ICLR), 2024. arXiv:2310.06770.
- [4] J. Yang et al., "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," Advances in Neural Information Processing Systems (NeurIPS), 2024. arXiv:2405.15793.
- [5] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafraan, K. R. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," ICLR, 2023.

- [6] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language Agents with Verbal Reinforcement Learning," NeurIPS, 2023. arXiv:2303.11366.
- [7] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching Large Language Models to Self-Debug," ICLR, 2024. arXiv:2304.05128.
- [8] S. Hong et al., "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework," ICLR, 2024. arXiv:2308.00352.
- [9] C. Qian et al., "ChatDev: Communicative Agents for Software Development," Proceedings of ACL, 2024. arXiv:2307.07924.
- [10] X. Wang et al., "OpenHands: An Open Platform for AI Software Developers as Generalist Agents," arXiv:2407.16741, 2024.
- [11] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous Program Improvement," Proceedings of ISSTA, 2024. arXiv:2404.05427.
- [12] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," NeurIPS, 2023. arXiv:2305.01210.
- [13] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based Software Engineering Agents," arXiv:2407.01489, 2024.
- [14] B. Roziere et al., "Code Llama: Open Foundation Models for Code," arXiv:2308.12950, 2023.
- [15] T. Schick et al., "Toolformer: Language Models Can Teach Themselves to Use Tools," NeurIPS, 2023. arXiv:2302.04761.
- [16] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation," IEEE Transactions on Software Engineering, vol. 50, no. 1, 2024. arXiv:2302.06527.
- [17] N. Jain et al., "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code," arXiv:2403.07974, 2024.
- [18] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, vol. 33, 2020.