International Journal of Modern Computer Science and IT Innovations (IJMCSIT)
eISSN: 3087-4289
Volume03 Issue011 | Page No: - 14-26

Received: 07 Oct 2026, Revised: 23 Oct 2026, Accepted: 03 Nov 2026, Published: 12 Nov 2026

DOI: https://doi.org/10.55640/ijmcsit-v03i11-02

Methods of Intelligent Automation of Software System Analysis Using AI Agents in the Development and Operation of Web Applications

Hovar Kiryl
Senior Full Stack Engineer Warsaw, Poland

ABSTRACT: AI agents built on large language models now take part in the analysis of software systems, yet evidence on their use across the life cycle of web applications remains scattered among benchmark reports and separate field studies. This article aims to systematize methods of agent-based analysis in the development and operation of web applications and to substantiate a model for assigning agent autonomy. The study combines a structured review of 25 sources published from 2020 to 2026, comparative analysis of benchmark results, content analysis of field studies, and a qualitative case study of two web products from the author's practice. The synthesis suggests that the strength of the verification oracle may be a more stable basis for assigning autonomy than benchmark capability scores; a direct comparative test of this hypothesis remains necessary. The article proposes an oracle-gated model with four layers, a matrix of four autonomy modes, and five monitoring indicators. The conclusions support staged adoption that begins with tasks backed by executable checks. The findings will interest web developers, game developers, engineering managers, and researchers in software engineering.

KEYWORDS: AI agents, large language models, software system analysis, web applications, software life cycle, automated testing, root cause analysis, verification oracle, human oversight, game development.

Introduction

Software analysis is understood here as the set of activities by which engineers establish what a system does, where it fails, and whether a change is safe. In web application projects these activities run from the first reading of an issue to the diagnosis of an incident in production. Early tools built on large language models assisted with code completion. Between 2024 and 2026 models gained the ability to plan, call tools, run tests, and read the results, which turned them into agents. Sajadieh et al. [1] report that the success rate of agents on WebArena, a benchmark built on functional websites, rose from about 15% in 2023 to 74.3% in early 2026, within four percentage points of the human baseline of 78.2%, and that the first-ranked model resolved approximately 76.8% of the tasks in SWE-bench Verified as of February 2026. The same report notes that agents still fail roughly one in three attempts on structured benchmarks [1].

Adoption among developers has followed. In the 2025 Developer Survey by Stack Overflow [2], which collected more than 49,000 responses from 177 countries, 84% of respondents used or planned to use AI tools in their development process, compared with 76% a year earlier, and 51% of professional developers used such tools daily. Trust moved in the opposite direction: 46% of respondents distrusted the accuracy of AI output and 33% trusted it. Agents were less common than assistants, with 14.1% of respondents using agents at work daily and 37.9% reporting no plans to adopt them [2]. At the level of organizations, Tinkoff et al. [3] found in a survey of 1,719 participants from 97 nations that 40% of respondents from companies with annual revenue above $1 billion were scaling AI agents in 2026, up from 27% a year earlier. Nearly one third of respondents (32%) said their organizations had decided against buying a software product or feature because the functionality could be built in house with agentic coding tools [3]. This points to a growing volume of internally produced code

and, with it, a growing demand for analysis. A field experiment by Cui et al. [4] across three companies and 4,867 developers measured a 26.08% increase in completed tasks among developers who had access to an AI coding assistant. Taken together, these data describe a technology whose use grows faster than confidence in its output.

The research literature has grown at a comparable pace. Hou et al. [5] reviewed 395 papers on large language models in software engineering published from January 2017 to January 2024. He et al. [6] examined multi-agent systems across the phases of the software life cycle and set out a research agenda toward what they call Software Engineering 2.0. Both reviews are organized around model capabilities and task types, and they leave three matters open. Published evidence is tied to separate tasks, such as issue resolution or incident diagnosis, and is rarely read as one chain that runs from a commit to production monitoring. The benchmarks in common use are built on Python repositories, while a web application adds a browser, visual output, and user data. The literature also offers little guidance on how much autonomy an agent should receive for a given analysis task, which is the decision a practicing web developer faces daily.

The aim of this study is to systematize methods of intelligent automation of software system analysis with AI agents across the development and operation of web applications and to substantiate a model for assigning agent autonomy to analysis tasks. **The author's hypothesis** is that the reliability of agent-based analysis depends more on the strength of the verification oracle attached to a task, that is, the check that decides whether an output is correct, than on the degree of autonomy granted to the agent. The present synthesis motivates this hypothesis but does not isolate these effects experimentally; autonomy should therefore rise only with stronger, independently assessed checks. **The scientific novelty** lies in the systematization of agent-based analysis methods along the full life cycle of a web application and in a conceptual model that links agent autonomy to oracle strength and to the cost of an undetected error, illustrated on two classes of web products.

The study has limits that follow from its design. It relies on published benchmark results and field studies and does not include a controlled experiment of its own. The benchmarks cited use different task sets, so their figures are compared as orders of magnitude and not as points on one scale. The two products from the author's practice are described qualitatively. The field also changes quickly, and the numerical values reflect the evidence available in the first half of 2026. Accordingly, the model is a conceptual proposal rather than an experimentally validated causal rule; its assignments should be evaluated within each product and workflow.

The practical value of the work lies in the gap between investment in AI and its routine use. Private investment in AI reached record levels in 2025, yet the deployment of agents stayed in single digits in nearly all business functions of the organizations surveyed [1]. That gap is closed in practice by engineers who turn agent capabilities into dependable procedures. Small product teams, providers of software for legal services, and studios in interactive entertainment need such procedures in a form they can apply without a research department, and the model proposed here is designed for them. The sources and procedures used to examine the hypothesis are described next.

Materials and methods

The practical part uses two products from the author's practice: a web application that displays incident events on a map for attorneys, and a browser-based tactics game launched from a messenger. Both are described at the level of their functional class. No internal metrics of these products are reported; they are the objects to which the model is applied. The results below move from the published evidence to the model and from the model to its application

Results and discussion

Read as one chain from issue intake to production monitoring, the sources yield eleven classes of agent-based analysis. A table suits this material because every class has to be compared on the same attributes: the task and its object, the method, the reported result, and the check that decides success. Table 1 gives this comparison.

Table 1. Classes of agent-based analysis across the life cycle of a web application (compiled by the author based on [1, 7-12, 14-16, 19-21, 24])

Analysis task and object	**Method class**	**Reported result**	**Check that decides success**
Issue intake: issue text and repository	Retrieval of relevant files followed by one patch attempt	1.96% of 2,294 issues from 12 Python repositories resolved	Repository tests
Fault localization: classes and methods in the syntax tree	Structure-aware code search, with spectrum-based localization when tests exist	19% of 300 issues in SWE-bench Lite at an average cost of $0.43	Repository tests
Repair with a fixed pipeline: suspect files and candidate patches	Localization, repair, and patch validation with no actions chosen by the model	32.00% of SWE-bench Lite (96 fixes) at $0.70 per issue	Repository tests and patch validation
Repair with an interactive agent: files, shell, test runner	Agent-computer interface for editing, navigation, and execution	12.5% of SWE-bench and 87.7% of HumanEvalFix	Repository tests
Front-end and visual defects: JavaScript libraries and issue images	Multimodal issue resolution	12% of 617 tasks from 17 JavaScript libraries; 6% for the next system	Tests with visual elements
Unit testing: API functions and existing test classes	Prompted generation with repair of failing tests; filtered improvement of tests written by people	Median statement coverage of 70.2% on 25 npm packages; 75% of test cases built, 57% passed reliably, 25% raised coverage	Build, repeated pass, coverage gain
Code review: pull request changes	Review comments generated by a language model	73.8% of comments marked resolved; average closure time rose from 5 h 52 min to 8 h 20 min	Developer decision on each comment
Behavior in the browser: running web application	Web navigation agent	14.41% of tasks in 2023 and 74.3% in early 2026; human participants 78.24%	State of the site after the task
Security of generated code: completed programs	Scenario-based assessment against known weakness classes	About 40% of 1,689 programs vulnerable across 89 scenarios	Weakness classes from the MITRE Top 25 list

Analysis task and object	Method class	Reported result	Check that decides success
Incident diagnosis: alerts, logs, diagnostic data	Handler matching with root cause category prediction; agent with retrieval tools	Root cause accuracy up to 0.766 on one year of incidents; agent competitive with baselines at higher factual accuracy	Confirmed root causes of past incidents
Game quality assurance: rendered frames	Multimodal glitch detection	43.4% for the first-ranked of 11 models on 593 glitches from 205 games	Glitches labelled by people

Two regularities follow from Table 1. The first concerns the relation between autonomy and outcome; the second, discussed after Figure 1, concerns the nature of the check. Xia et al. [10] built a pipeline of three fixed phases in which the model never chooses its next action, and on SWE-bench Lite it exceeded every open-source agent compared in that study at a cost of $0.70 per issue. Zhang et al. [9] reached 19% on the same subset by searching the program structure instead of treating the project as a collection of files, at an average cost of $0.43. In both cases the gain came from better evidence and a stricter check, and the freedom of the agent was reduced.

Change over time is easier to see in a chart than in a table, because the values span two orders of magnitude. Figure 1 places the issue resolution rates reported in the sources next to the task completion rates in WebArena.

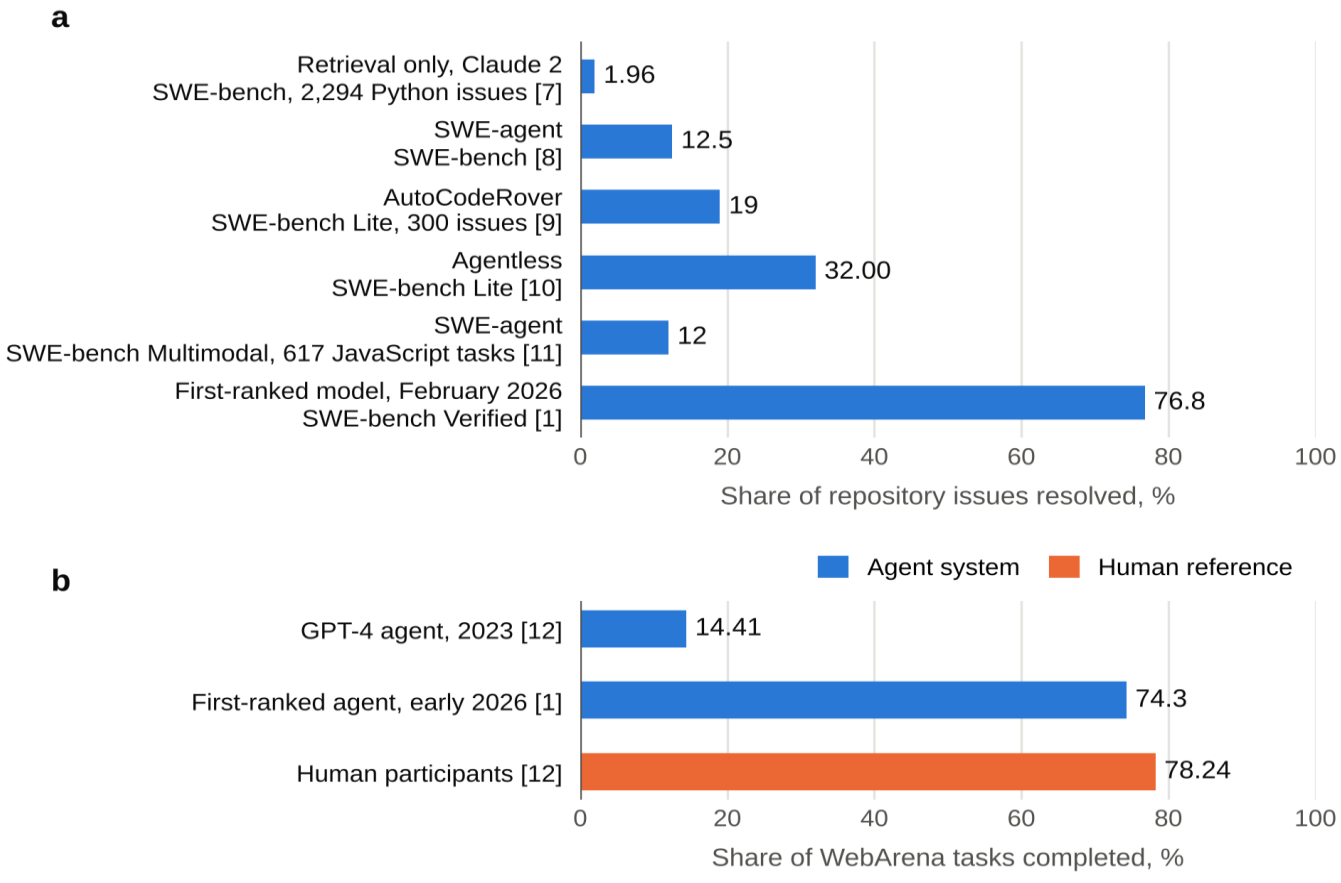


Figure 1. Reported success rates of agent systems in repository issue resolution (a) and in WebArena web tasks (b), % (compiled by the author based on [1, 7-12])

Panel (a) combines results obtained on different subsets of SWE-bench, so it should be read as a direction of change; the bars do not form a single trend line. In under three years the reported share of resolved issues moved from 1.96% with retrieval alone [7] to approximately 76.8% on the human-validated subset [1]. Panel (b) shows a comparable change for tasks carried out in a browser, where the gap to human participants narrowed from 64 percentage points to four [1, 12]. One bar departs from this pattern. On JavaScript tasks with visual content the first-ranked system resolved 12% of the instances [11]. The libraries in that benchmark serve web interface design, diagramming, data visualization, syntax highlighting, and interactive mapping, which are common components of a web front end. Progress is therefore uneven: fast where a test suite exists and the task is stated in text, slow where the result has to be judged by its appearance. For a web developer this means that an agent can be trusted with different degrees of independence in different parts of the same product.

The second regularity in Table 1 concerns the nature of the check. Where success is decided by executable tests, results are reported as shares of tasks solved and can be compared across studies. Where success is decided by a person, as in code review, the acceptance of agent output was high (73.8% of comments resolved), and the process as a whole slowed [16]. The breadth of the research base supports the same reading: Wang et al. [13] analyzed 102 studies on language models in software testing, and Zhang et al. [22] analyzed 183 papers on language models in IT operations published from January 2020 to December 2024, yet in the sources reviewed here results for operation are reported on proprietary incident data [20, 21], which makes them harder to compare than results for repair.

Benchmarks describe what agents can do under controlled conditions, while field studies show what happens when people work with them. The field studies differ in design and sample, and a table that keeps these attributes side by side prevents their numbers from being read as one series. Table 2 summarizes the field evidence.

Table 2. Field evidence on AI assistants and agents in software teams (compiled by the author based on [2-4, 15-18, 20])

Setting and sample	**Measured outcome**
Randomized trials at Microsoft, Accenture, and a Fortune 100 company; 4,867 developers	26.08% more completed tasks (standard error 10.3%); higher adoption and larger gains among less experienced developers
Survey of GitHub Copilot users matched with usage data	The rate at which shown suggestions are accepted drives perceived productivity more than the persistence of accepted code
Survey of 410 developers	Motives for use: fewer keystrokes, faster completion, recall of syntax. Reasons for non-use: output misses functional or non-functional requirements; output is hard to control
Test-a-thons for Instagram and Facebook at Meta	11.5% of the classes to which the tool was applied were improved; 73% of recommendations accepted for production
238 practitioners in ten projects; 4,335 pull requests, of which 1,568 received automated review	73.8% of comments resolved; average closure time up from 5 h 52 min to 8 h 20 min; a minor improvement in code quality reported by a majority of practitioners
One year of cloud incidents at Microsoft	Root cause accuracy up to 0.766; the diagnostic collection component in use for over four years

Setting and sample	Measured outcome
More than 49,000 responses from 177 countries	69% of agent users report higher productivity and 37.5% report better code quality; 86.9% are concerned about the accuracy of information from agents
1,719 participants from 97 nations	40% of respondents from large organizations scale agents, up from 27% a year earlier; scaling is reported more often in IT, knowledge management, and software engineering than in other functions

Table 2 shows an asymmetry. Indicators of volume and speed respond readily: more tasks are completed [4], suggestions are accepted [17], and 69% of agent users report higher productivity [2]. Indicators of quality respond weakly or in the opposite direction. Only 37.5% of agent users report better code [2], pull requests take longer to close [16], and a laboratory assessment found weaknesses in about 40% of generated programs [19]. The deployments with a clear quality gain are those in which every agent output passes a filter before a person sees it. The tool described by Alshahwan et al. [15] kept only the tests that were built, passed repeatedly, and raised coverage, and 73% of its recommendations were then accepted. The system described by Chen et al. [20] works on diagnostic data gathered by handlers that engineers had maintained for over four years. One further detail in Table 2 bears on the hypothesis. Cui et al. [4] found higher adoption and larger gains among less experienced developers, a group with less accumulated knowledge for judging a generated change. The more such developers rely on agents, the more the correctness of the result depends on checks that do not require their judgment.

In the author's reading, these observations support the hypothesis. The constraint in agent-assisted work has moved from producing changes to establishing that they are correct, and the return on agents is set by the verification capacity that surrounds them. The reasons developers give for not using assistants point the same way: output that misses requirements and output that is hard to control [18] both concern checking more than generation. The gap between the autonomy given to an agent and the strength of the check available for its output is called here the oracle deficit. A team that runs agents with a deficit pays for it later in review time and in defects that reach users, much as a team with technical debt pays in maintenance. The model proposed below seeks to reduce this deficit by matching autonomy to the available evidence and independent checks, while leaving residual risk subject to human review.

The evidence can be organized into a model with four layers: evidence, agents, oracles, and decision. The model differs from the pipelines described in the reviewed literature in one respect. The oracle layer is placed between the agents and the developer as a separate layer, and outputs with executable or partial oracles carry a gate verdict to the decision layer, whereas M1 findings without an executable oracle are routed directly to developer review with their supporting evidence. This placement applies the position of Shneiderman [25] that high levels of human control and high levels of automation can be designed to coexist. The developer keeps control because the gate determines what is presented for decision. A diagram is used because the model is defined by the direction of the flows between layers, which a list would hide. Figure 2 shows the layers and their links.

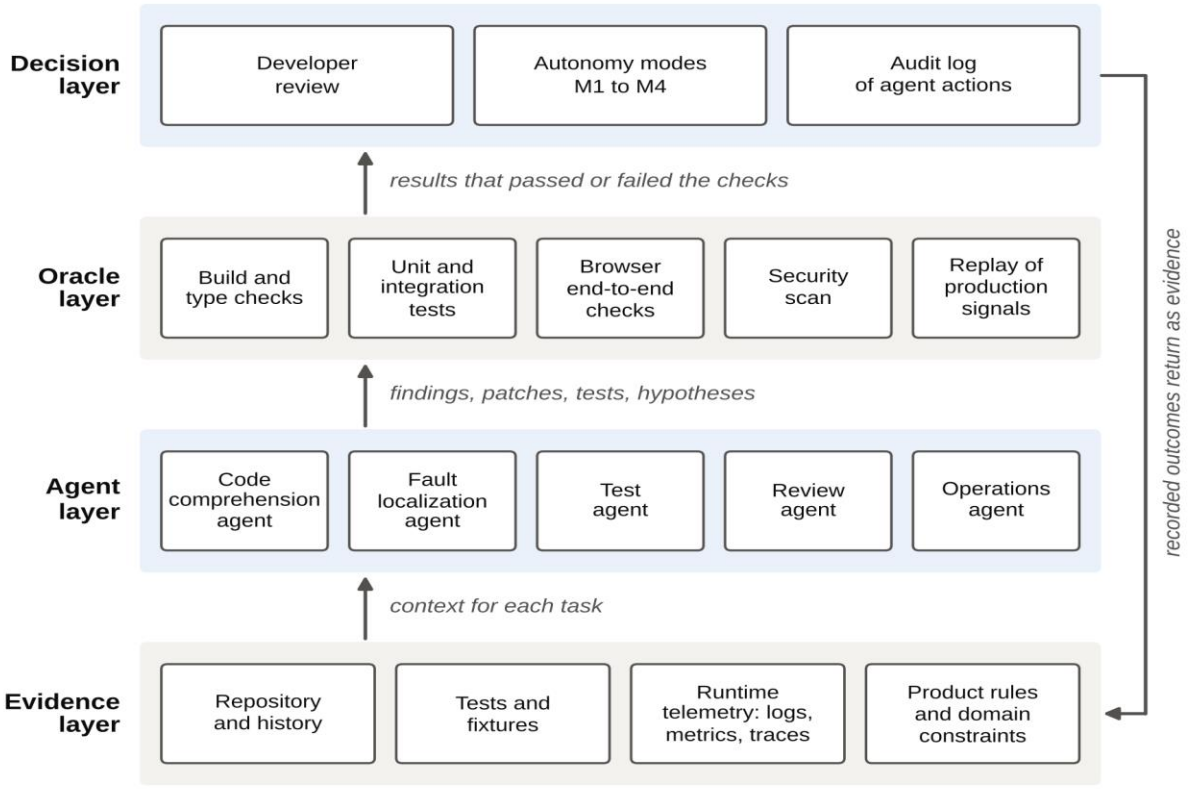


Figure 2. Layered model of oracle-gated agent-based analysis for a web application (compiled by the author based on [6, 9, 15, 20, 21, 25])

The evidence layer holds everything an agent may read: the repository with its history, tests and fixtures, runtime telemetry, and product rules. The sources indicate that the form of this evidence matters as much as its volume. Structure-aware search over classes and methods improved localization [9], and Roy et al. [21] show that an agent for incident diagnosis needs the ability to collect additional logs and metrics during its work, which earlier approaches lacked. For a web application the telemetry includes errors raised in the browser as well as server logs.

The agent layer contains five roles defined by the analysis function: code comprehension, fault localization, testing, review, and operations. The roles are separated because they read different evidence and answer to different checks, in line with the cooperative arrangements reviewed by He et al. [6]; a single underlying model may serve all of them. The oracle layer consists of checks that return a verdict without the participation of a language model: build and type checks, unit and integration tests, end-to-end checks in a browser, a security scan, and the replay of production signals against a candidate fix. The decision layer contains the developer, the autonomy mode assigned to the task, and an audit log of agent actions.

The model needs a rule for choosing the mode in the decision layer. Two properties of a task are enough to state the rule, and a matrix shows all their combinations at once. Figure 3 presents the autonomy assignment matrix.

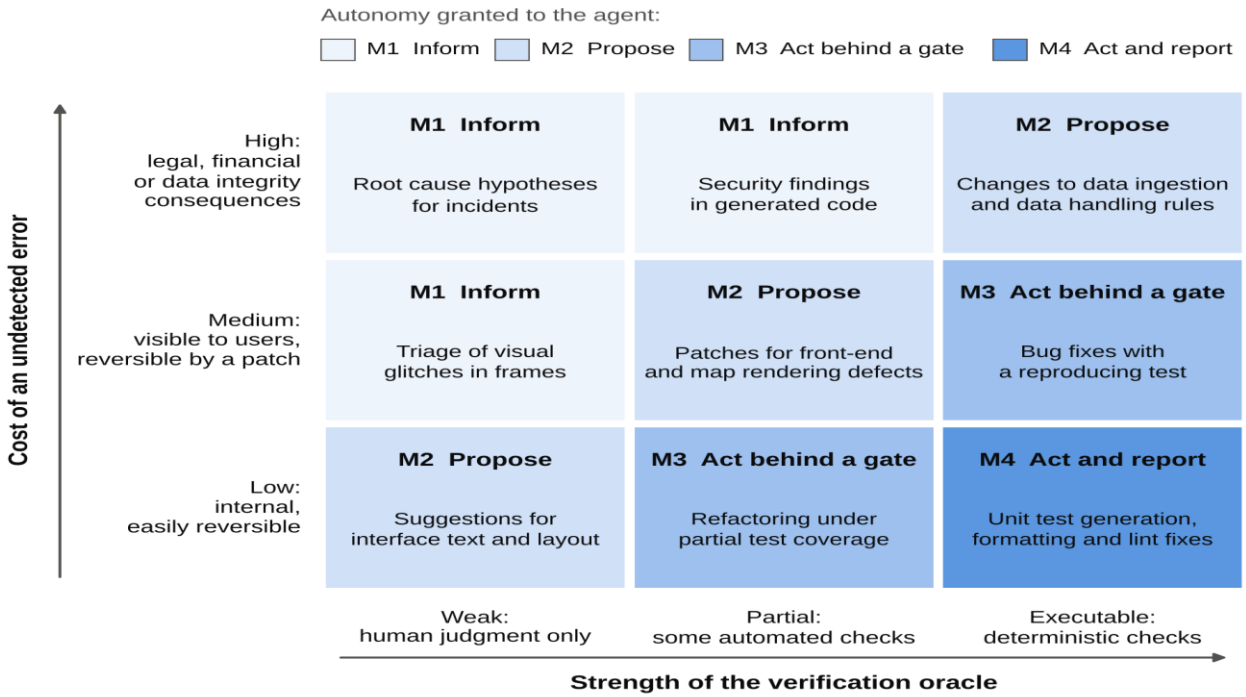


Figure 3. Matrix for assigning agent autonomy by oracle strength and by the cost of an undetected error (compiled by the author based on [11, 15, 19, 21, 24, 25])

The matrix defines four modes. In M1 (Inform) the agent reports findings and changes nothing. In M2 (Propose) it prepares a change or a hypothesis, and a developer approves each one. In M3 (Act behind a gate) a change is applied automatically when the oracle passes, and a developer reviews a sample afterwards. In M4 (Act and report) the agent acts within a bounded scope that can be rolled back, and the developer reads summaries. The rule can be stated in one line. With oracle strength and error cost each scored from 0 to 2, the mode number equals 2 plus the oracle score minus the cost score, bounded between 1 and 4. Each step of oracle strength raises autonomy by one mode, and each step of error cost lowers it by one.

The placement of example tasks follows from the two axes. Triage of visual glitches has only human judgment as its oracle and a medium cost of error, which gives M1; the benchmark score of 43.4% for the first-ranked multimodal model [24] illustrates the limitations of current visual detection, but does not determine the mode. Patches for front-end defects have a partial oracle and a medium cost, which gives M2, consistent with the 12% of visual JavaScript tasks resolved in [11]. Unit test generation belongs to M4 when build, repeated-pass, and coverage-gain filters of the kind described in [15] provide an executable oracle and faulty test code is confined to CI rather than reaching users. Security findings in generated code stay in M1 because functional tests do not reliably detect weaknesses of the kind reported by Pearce et al. [19], and a missed weakness carries a high cost. Root cause hypotheses stay in M1 because their confirmation rests with on-call engineers who know the affected services [21].

Agent capability is deliberately absent from the axes. Capability changes within months, as Figure 1 shows, while oracle strength and error cost are properties of the product that a team controls and can measure. In the model a gain in capability appears as a higher pass rate at the gate, while the mode stays the same. A team that wants more autonomy for a class of tasks obtains it by strengthening the oracle for that class.

The order of operations for a single task is shown as a flowchart, since it contains two decision points, an M1 bypass for human-only judgment, and bounded return paths. The loop accepts triggers from both phases of the life cycle: a commit or an issue during development, an alert or a scheduled scan during operation. After the evidence is assembled, the task is classified with the matrix before the agent starts, so the mode is fixed in advance and does not depend on how confident the output appears. Figure 4 gives the loop.

For M1, the output bypasses the executable-oracle gate and goes directly to developer review together

with the evidence used by the agent. For the other modes, bounded return paths limit failed checks and rework. A failed check returns to the agent together with the error, which is the mechanism Schäfer et al. [14] used when they prompted the model again with the failing test and its message, and which the interface described by Yang et al. [8] provides through execution feedback. The number of retries is fixed. The costs reported in the sources, $0.43 and $0.70 per issue [9, 10], indicate that a small number of attempts is affordable for a small team, and a limit prevents an agent from spending without progress. When the limit is reached, the task goes to the developer with the full trace. Every outcome, including a rejection, is written to the audit log and returned to the evidence layer, where it informs later tasks of the same class.

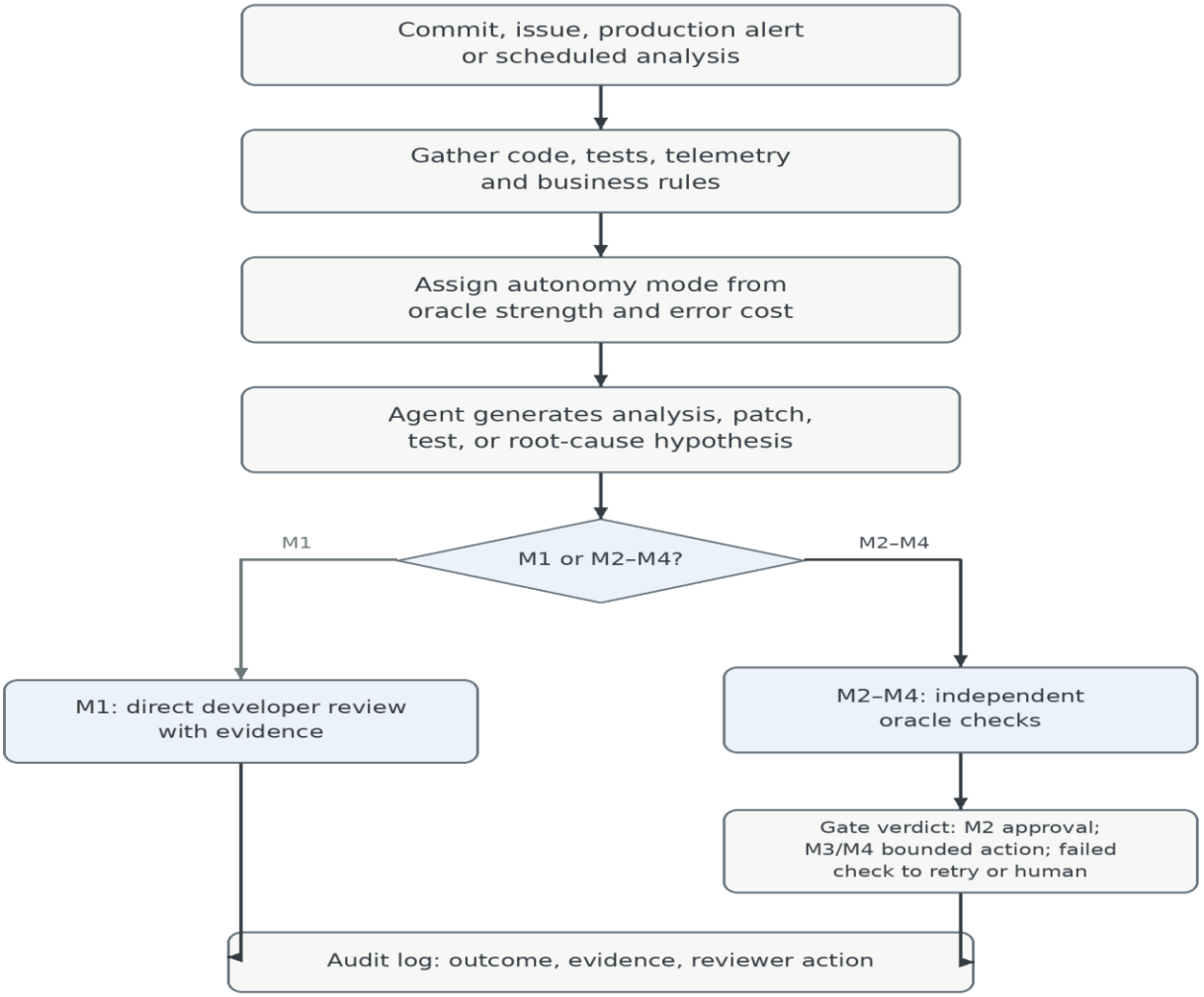


Figure 4. Flow of a single analysis task through the oracle-gated loop (compiled by the author based on [8-10, 14, 15])

A short illustration shows how the loop treats one task; the scenario is constructed for explanation and reports no measurement. Suppose a player of the web game reports that a move was accepted although the rules forbid it. The evidence assembled for the task consists of the recorded session, the rule code, and the existing tests. The task is classified as medium in error cost, because players see the fault and a patch removes it, and as executable in oracle strength, because the session can be replayed with a fixed random seed. The matrix gives M3. The testing agent first writes a session that reproduces the fault, the localization agent proposes a patch, and the gate runs the new session together with all rule invariants. If the gate passes, the change is applied and the developer later reviews it as part of a sample. If the gate fails on every permitted attempt, the developer receives the trace. An identical report about a wrong event location in the map-based application would be classified as high in error cost and would stop at M2, with a person approving the correction.

To check that the model discriminates between products, it was applied to two web products from the author's practice that differ in the cost of an error. A table

is used because the same dimensions are read for both products. Table 3 gives the resulting profiles.

Table 3. Profiles of two web products under the oracle-gated model (compiled by the author based on [1, 11, 12, 20, 23, 24])

Dimension	**Map-based incident application for attorneys**	**Browser-based tactics game in a messenger**
Function	Displays incident events on a map for use by attorneys	Game opened from a Telegram bot and played in a web view
Objects of analysis	Ingestion of events and their records; map rendering; filters; rules for handling event data	Game rules and state transitions; client rendering; start of a session inside the messenger
Defect classes typical of the product type	Wrong place or time of an event; duplicates; stale data; faults in map layers	Rule violations; mismatch between displayed and actual game state; balance regressions; visual glitches
Available oracles	Schema and coordinate validation; comparison with source records; browser checks of filter results	Scripted play sessions with fixed random seeds; rule invariants; screenshot comparison
Cost of an undetected error	High: a professional user may rely on a wrong or missing event	Low to medium: player experience is affected and a patch restores it
Assigned modes	M1 for incident hypotheses; M2 for ingestion, data handling rules, and map rendering; M3 for test code, where a faulty test fails the CI gate and does not reach users	M4 for unit tests; M3 for fixes with a reproducing session; M2 for rule changes; M1 for glitch triage
Priority agent roles	Review and operations	Testing and code comprehension
Indicators to track	Share of agent findings confirmed by an oracle; time from alert to confirmed cause; data defects found after release	Share of rules covered by invariants; pass rate at the gate on the first attempt; glitch reports per release

The profile of the map-based application is conservative. Its users are attorneys, for whom a wrong location or a missing event has professional consequences, so each proposed change to the underlying event data remains subject to M2 human approval even where tests exist. Test code itself can use M3 when an executable CI gate prevents a faulty test from being shipped. The evidence agrees with this placement: interactive mapping libraries are among those on which the first-ranked system resolved 12% of the tasks [11], and root cause accuracy of up to 0.766 in a mature industrial setting [20] means that roughly one predicted root cause in four is wrong. Agents remain useful here in a different capacity. Since agents now complete 74.3% of WebArena tasks, and that environment includes a map tool [1, 12], a web navigation agent can run filter scenarios in a browser and

hand the resulting site state to a deterministic comparison. In this arrangement the agent operates the product and the oracle judges the result.

The profile of the game is permissive. Rules can be expressed as invariants and exercised by scripted play, which gives an executable oracle for a large part of the logic, and an error is corrected by the next release. Gallotta et al. [23] list the player and the analyst among the roles a language model can take in a game, and both roles fit the loop in Figure 4: an agent that plays scripted sessions produces traces, and the testing agent turns those traces into rule checks. Visual quality is the exception. Frame analysis stays in M1 because its correctness oracle is human visual judgment; the 43.4% benchmark result [24] illustrates the remaining recognition challenge rather than assigning the mode.

The same model thus yields two different profiles, and the difference is traceable to two inputs that a developer can state before any agent is run. This is the property that makes the model usable by a solo developer who maintains products of both kinds.

The synthesis leads to six recommendations for teams that develop and operate web applications. They are ordered by the sequence in which a team would apply them.

First, build the oracle before granting autonomy. For each class of analysis task, a team should record what decides success. Where nothing executable exists, the first assignment for an agent is to help write the check, working in M3 under build and pass filters of the kind described in [15].

Second, assign modes with the matrix and review the assignments on a fixed schedule, for example once a quarter and after every incident. The review asks two questions only: whether the oracle for the class has changed and whether the cost of an error has changed.

Third, track the outcomes of analysis instead of the volume of output. Five indicators are proposed: the share of agent findings confirmed by an oracle or by a developer; the pass rate at the gate on the first attempt; review latency, measured from agent output to developer decision; the number of defects reported after each release; and the time from alert to confirmed cause. The first two measure the agent, the third measures the load on people, and the remaining two measure the product. Acceptance rates alone are a weak guide, since they track perceived productivity [17] and can coexist with slower delivery [16].

Fourth, keep the security check independent of the agent that wrote or reviewed the change, and run it on every change regardless of mode [19].

Fifth, limit what enters the evidence layer. In the developer survey 81.4% of respondents to the corresponding survey question expressed concern about the security and privacy of data when using agents [2]. For products that hold client data, such as software for legal practice, the default is that production records are excluded from agent context and replaced with synthetic fixtures.

Sixth, small teams should begin with two low-risk entries: operations in M1, where the agent drafts incident summaries and hypotheses, and isolated test-code changes in M3 where CI provides an executable gate and user-facing behavior is unaffected. The audit log produced by the loop then doubles as documentation of how each change was verified, which has value for clients in regulated fields. These recommendations presuppose version control, continuous integration, and basic telemetry; where these are absent, establishing them comes before any use of agents. The conclusion returns to the aim and the hypothesis stated in the introduction.

A direct validation of the autonomy-assignment hypothesis remains future work. A 2 × 2 design would apply the same model to a fixed set of previously resolved issues from each product, varying autonomy (fixed pipeline versus autonomous agent) and oracle strength (existing tests versus tests supplemented with invariants, replayed sessions, or browser checks). The assessment would compare accepted-fix rate, escape rate among gate-passing outputs, first-attempt pass rate, review time, and cost per issue. No results from this proposed experiment are claimed in the present study.

Conclusion

The study set out to systematize agent-based methods for the analysis of software systems across the development and operation of web applications and to substantiate a model for assigning agent autonomy. Both parts of the aim were met. The sources were organized into eleven classes of analysis, from issue intake to game quality assurance, each described by its object, method,

reported result, and deciding check.

The hypothesis is compatible with the evidence but is not directly established by it. A pipeline with fixed phases and patch validation exceeded the open-source agents it was compared with, test generation under build, pass, and coverage filters had 73% of its recommendations accepted for production, and automated review without an executable check lengthened pull request closure from 5 h 52 min to 8 h 20 min. Field data show the same asymmetry between speed and quality: 26.08% more completed tasks in a randomized trial, against 37.5% of agent users who report better code.

The proposed oracle-gated model has four layers (evidence, agents, oracles, decision), four autonomy modes selected by oracle strength and error cost, a loop with a fixed retry limit and an audit log, and five monitoring indicators. Applied to a map-based incident application for attorneys and to a web game in Telegram, it produced a conservative profile for the first and a permissive profile for the second, with the difference explained by two inputs.

The practical significance of the work is that a web developer or a small studio can decide, before running any agent, which analysis tasks to delegate and under which checks, and can raise autonomy by strengthening tests instead of waiting for stronger models. A reasonable next step is to measure the five indicators on live projects over several release cycles, beginning with the two products described here.

References

1. Sajadieh, S., Fattorini, L., Perrault, R., Gil, Y., Parli, V., Santarlasci, L., Pava, J., Maslej, N., Altman, R., Brynjolfsson, E., Brodley, C., Clark, J., Dignum, V., Kumar, V., Landay, J., Lyons, T., Manyika, J., Niebles, J. C., Shoham, Y., . . . Weld, D. (2026). *Artificial Intelligence Index Report 2026*. Stanford Institute for Human-Centered Artificial Intelligence. https://hai.stanford.edu/ai-index/2026-ai-index-report
2. Stack Overflow. (2025). *2025 Developer Survey*. https://survey.stackoverflow.co/2025/
3. Tinkoff, D., Van der Veken, L., Chui, M., & Balakrishnan, T. (2026, August 25). *The state of AI in 2026: On the road to ROI*. McKinsey & Company. https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai
4. Cui, K. Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2026). The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. *Management Science*. Advance online publication. https://doi.org/10.1287/mnsc.2025.00535
5. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology, 33*(8), 1-79. https://doi.org/10.1145/3695988
6. He, J., Treude, C., & Lo, D. (2025). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology, 34*(5), 1-30. https://doi.org/10.1145/3712003
7. Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., & Narasimhan, K. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html
8. Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems 37* (pp. 50528-50652). https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html
9. Zhang, Y., Ruan, H., Fan, Z., & Roychoudhury, A. (2024). AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 1592-1604). ACM. https://doi.org/10.1145/3650212.3680384
10. Xia, C. S., Deng, Y., Dunn, S., & Zhang, L. (2025). Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering, 2*(FSE), 801-824. https://doi.org/10.1145/3715754
11. Yang, J., Jimenez, C. E., Zhang, A. L., Lieret, K., Yang, J., Wu, X., Press, O., Muennighoff, N., Synnaeve, G., Narasimhan, K. R., Yang, D., Wang, S. I., & Press, O. (2025). SWE-bench Multimodal: Do AI

systems generalize to visual software domains? In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR 2025)*. https://proceedings.iclr.cc/paper_files/paper/2025/hash/07d6332ae36730707fddddba736d7b6c-Abstract-Conference.html

12. Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., & Neubig, G. (2024). WebArena: A realistic web environment for building autonomous agents. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*. https://proceedings.iclr.cc/paper_files/paper/2024/hash/4410c0711e9154a7a2d26f9b3816d1ef-Abstract-Conference.html
13. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering, 50*(4), 911-936. https://doi.org/10.1109/TSE.2024.3368208
14. Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering, 50*(1), 85-105. https://doi.org/10.1109/TSE.2023.3334955
15. Alshahwan, N., Chheda, J., Finogenova, A., Gokkaya, B., Harman, M., Harper, I., Marginean, A., Sengupta, S., & Wang, E. (2024). Automated unit test improvement using large language models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (pp. 185-196). ACM. https://doi.org/10.1145/3663529.3663839
16. Cihan, U., Haratian, V., İçöz, A., Gül, M. K., Devran, Ö., Bayendur, E. F., Uçar, B. M., & Tüzün, E. (2025). Automated code review in practice. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 425-436). IEEE. https://doi.org/10.1109/ICSE-SEIP66354.2025.00043
17. Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2024). Measuring GitHub Copilot's impact on productivity. *Communications of the ACM, 67*(3), 54-63. https://doi.org/10.1145/3633453
18. Liang, J. T., Yang, C., & Myers, B. A. (2024). A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (pp. 1-13). ACM. https://doi.org/10.1145/3597503.3608128
19. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2025). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Communications of the ACM, 68*(2), 96-105. https://doi.org/10.1145/3610721
20. Chen, Y., Xie, H., Ma, M., Kang, Y., Gao, X., Shi, L., Cao, Y., Gao, X., Fan, H., Wen, M., Zeng, J., Ghosh, S., Zhang, X., Zhang, C., Lin, Q., Rajmohan, S., Zhang, D., & Xu, T. (2024). Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems* (pp. 674-688). ACM. https://doi.org/10.1145/3627703.3629553
21. Roy, D., Zhang, X., Bhave, R., Bansal, C., Las-Casas, P., Fonseca, R., & Rajmohan, S. (2024). Exploring LLM-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (pp. 208-219). ACM. https://doi.org/10.1145/3663529.3663841
22. Zhang, L., Jia, T., Jia, M., Wu, Y., Liu, A., Yang, Y., Wu, Z., Hu, X., Yu, P. S., & Li, Y. (2026). A survey of AIOps in the era of large language models. *ACM Computing Surveys, 58*(2), 1-35. https://doi.org/10.1145/3746635
23. Gallotta, R., Todd, G., Zammit, M., Earle, S., Liapis, A., Togelius, J., & Yannakakis, G. N. (2024). Large language models and games: A survey and roadmap. *IEEE Transactions on Games*. Advance online publication. https://doi.org/10.1109/TG.2024.3461510
24. Taesiri, M. R., Feng, T., Bezemer, C.-P., & Nguyen, A. (2024). GlitchBench: Can large multimodal models detect video game glitches? In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 22444-22455). IEEE. https://doi.org/10.1109/CVPR52733.2024.02118
25. Shneiderman, B. (2020). Human-centered artificial intelligence: Reliable, safe & trustworthy. *International Journal of Human-Computer Interaction, 36*(6), 495-504. https://doi.org/10.1080/10447318.2020.1741118