GBQA: A GAME BENCHMARK FOR EVALUATING LLMS AS QUALITY ASSURANCE ENGINEERS

Shufan Jiang[1,4] **Chios Chen**[2] **Zhiyang Chen**[†3]

[1]The University of Hong Kong [2]Independent Researcher [3]Westlake University [4]Datawhale Org.

ABSTRACT

The autonomous discovery of bugs remains a significant challenge in modern software development. Compared to code generation, the complexity of dynamic runtime environments makes bug discovery considerably harder for large language models (LLMs). In this paper, we take game development as a representative domain and introduce the Game Benchmark for Quality Assurance (GBQA), a benchmark containing 30 games and 124 human-verified bugs across three difficulty levels, to evaluate whether LLMs can autonomously detect software bugs. The benchmark is constructed using a multi-agent system that develops games and injects bugs in a scalable manner, with human experts in the loop to ensure correctness. Moreover, we provide a baseline interactive agent equipped with a multi-round ReAct loop and a memory mechanism, enabling long-horizon exploration of game environments for bug detection across different LLMs. Extensive experiments on frontier LLMs demonstrate that autonomous bug discovery remains highly challenging: the best-performing model, Claude-4.6-Opus in thinking mode, identifies only 48.39% of the verified bugs. We believe GBQA provides an adequate testbed and evaluation criterion, and that further progress on it will help close the gap in autonomous software engineering.

1 INTRODUCTION

Real-world software development is systematic: no non-trivial system is correct and robust on the first attempt, thus requiring an inherently iterative software engineering workflow. Traditionally, human developers follow repeated cycles of implementation, testing, debugging, and refactoring as shown in Figure 1(a). Currently, coding agents like Claude Code (Anthropic, 2025a), Cursor (Anysphere, 2024), and OpenAI Codex (OpenAI, 2025), actively participate in this development loop. In this paradigm, human developers provide natural language instructions, while LLMs generate code, execute the resulting program, inspect failures, and iteratively revise the code. This workflow, often referred to as vibe coding (Karpathy, 2025), pushes the frontier of automatic software engineering as illustrated in Figure 1(b) and (c).

Within this classical cycle, recent progress has dramatically strengthened the **development and fixing** side. Frontier LLMs can now generate project-level codebases from natural language specifications (Qian et al., 2024; Hong et al., 2024) and resolve real-world code issues given well-written bug reports or issue descriptions (Jimenez et al., 2024; Xia et al., 2024). However, the **testing and bug discovery** side of this loop remains largely unexplored, upon which the quality of a released software critically depends.

Unlike code generation or fixing, bug discovery poses fundamentally different challenges. First, the objective is ill-defined: the agent must proactively determine that "something is wrong" without being told what to look for, unlike generation or fixing tasks where a clear target or issue description is provided. Second, effective bug discovery demands comprehensive exploration and systematic

[†]Corresponding to volgachen@gmail.com

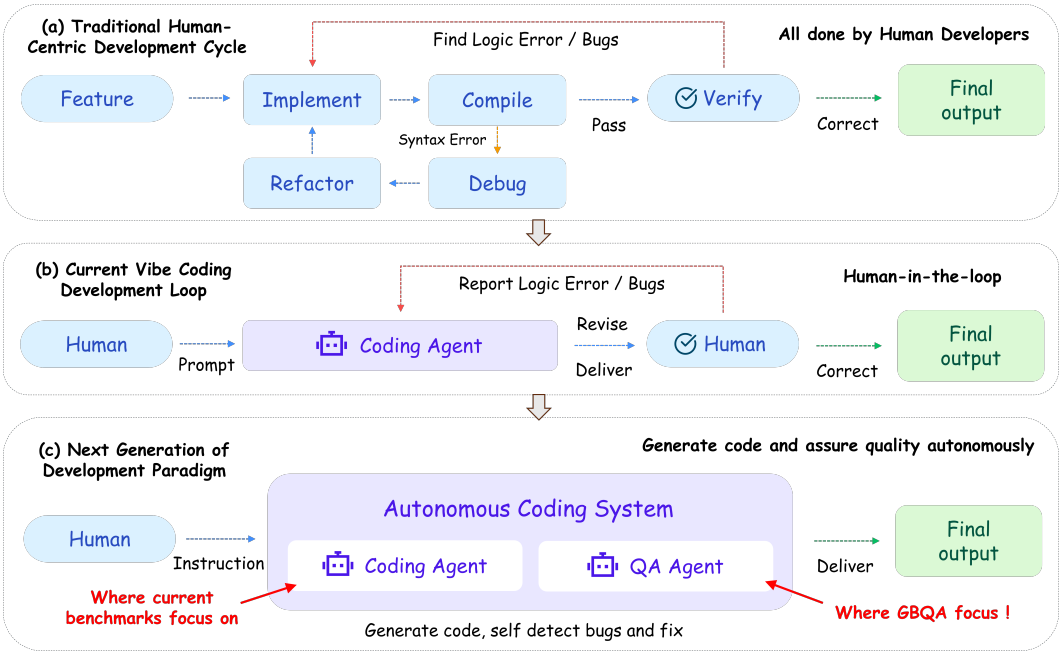


Figure 1: Evolution of the software development paradigm in the LLM era. (a) Traditional human-driven iterative workflow. (b) Human–LLM collaborative coding, where a coding agent assists development under human supervision. (c) Toward a fully autonomous coding system which can generate code, detect bugs and fix them without human-in-the-loop. While existing benchmarks primarily focus on code generation and fixing, our benchmark emphasizes autonomous bug discovery and quality assurance part within the development cycle.

planning over large behavioral state spaces, rather than targeted edits to a known location. Third, the agent must reason about the gap between expected and actual runtime behavior, often without access to explicit specifications. Most existing benchmarks bypass these difficulties by articulating a precise description in the task description before the agent intervenes. Consequently, the cognitively demanding work of perceiving anomalies and localizing their causes is still completed by humans. This upstream gap is similarly highlighted by recent efforts in autonomous code auditing (Guo et al., 2025b) and large-scale bug mining (Wu et al., 2025). Advancing toward fully autonomous system, therefore, requires directly evaluating and improving the ability of LLMs to discover defects independently.

In this paper, we take game development as the testbed for autonomous bug discovery. Games are self-contained software systems composed of internal state management, user input handling, and output rendering. They require long-term dynamic interactions within a single session, making them ideal representatives of real-world software engineering settings. At the same time, games expose clearly defined action spaces and state transitions, making agents easily construct formatted inputs and outputs, naturally compatible with agent-based exploration. Such interaction-driven, stateful verification is precisely the agentic capability that next-generation LLMs need to develop. Moreover, bug discovery in games corresponds to Quality Assurance (QA) in real world applications, which has a long tradition of systematic and specification-driven testing (Myers, 1979; Ammann & Offutt, 2016).

Motivated by these considerations, we introduce GBQA, a benchmark designed to evaluate the ability of LLMs to autonomously discover bugs in interactive game environments. As illustrated in Figure 2, GBQA contains 30 diverse games with a total of 124 human-verified bugs in them in order to evaluate how well an agent performs in the QA task. During evaluation, the agent autonomously explore the games, identify potential bugs, and report clear descriptions along with reproducible steps. Subsequently, each reported bug is then matched against the human-verified ground-truth annotations to compute quantitative metrics. The annotated bugs are categorized into different difficulty levels to assess model robustness across varying complexity. To construct this benchmark at scale, we

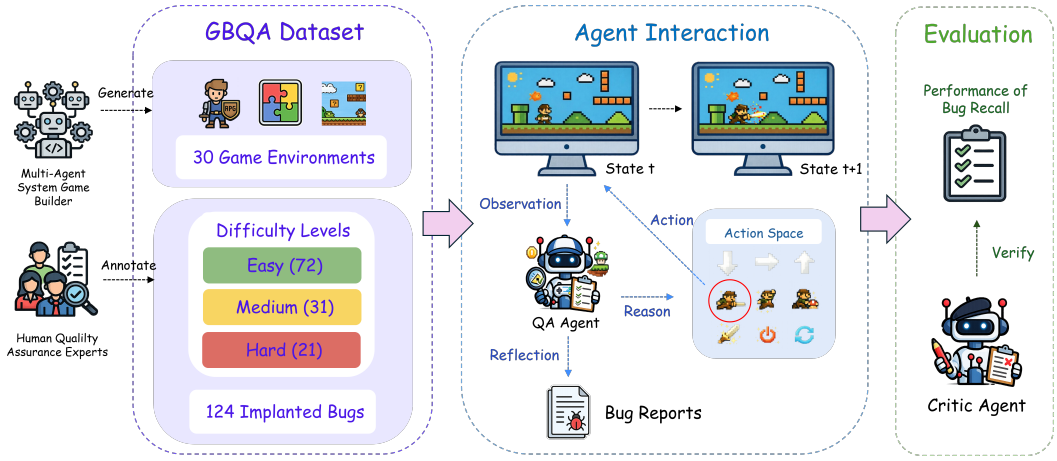


Figure 2: Overview of GBQA. Dataset is constructed using a multi-agent game builder that generates 30 game environments with 124 implanted bugs, which are annotated and categorized into three difficulty levels (Easy, Medium, Hard) by human QA experts. During evaluation, a QA agent autonomously interacts with the game environment through ReAct loops, and produces structured bug reports. Then, a critic agent verifies reported bugs by matching them against human-annotated ground truth to compute quantitative metrics.

develop a multi-agent system that automatically generates games and injects bugs with controllable complexity, while human experts remain in the loop to verify the correctness of all annotations.

To evaluate the capability of frontier LLMs in bug detection, we further provide a baseline interactive agent equipped with a multi-round ReAct loop and a memory mechanism, enabling long-horizon exploration of game environments. Our experiments demonstrate that autonomous bug discovery remains highly challenging: even the best-performing model, Claude-4.6-Opus in thinking mode, identifies less than half of the bugs, revealing substantial room for improvement.

Our contributions can be summarized as follows.

- We formalize the problem of autonomous bug discovery in interactive environments and present GBQA, a benchmark containing 30 diverse games and 124 human-verified bugs across three difficulty levels, along with a critic agent that supports automated evaluation.
- We develop a scalable game environment builder, including a multi-agent system capable of generating games and inserting bugs with controllable complexity, and introduce human-in-the-loop to ensure its correctness.
- We perform extensive evaluations of cutting-edge LLMs in GBQA, providing not only a comprehensive analysis of their performance and limitations but also a characterization of current failure modes in autonomous bug discovery.

2 Related Work

Software Engineering and Agent Benchmarks. Large language models have been widely evaluated on software engineering tasks. SWE-bench (Jimenez et al., 2024) and its extensions (Aleithan et al., 2024) measure an agent's ability to resolve real-world GitHub issues, while systems such as Agentless (Xia et al., 2024) improve issue localization and patch generation via structured pipelines. A shared assumption across these benchmarks is that the bug has already been identified and described by humans; agents are evaluated primarily on code repair. Beyond issue-driven repair, recent work explores automated defect detection in static repositories. RepoAudit (Guo et al., 2025b) and BugStone (Wu et al., 2025) analyze structural patterns and data dependencies to discover vulnerabilities at scale. However, these approaches operate on static code and do not assess the ability of agent to interact with a dynamic system, execute multi-step behaviors, and infer specification-level inconsistencies from runtime feedback. More generally, interactive agent benchmarks such

as WebArena (Zhou et al., 2024) and AgentBench (Liu et al., 2024) evaluate LLM agents in web navigation and tool-use scenarios. SMART (Mu et al., 2025) incorporates coverage-aware strategies for functional testing. In these settings, the environment is treated as ground truth and success is defined by task completion. In contrast, GBQA treats the environment itself as the object of evaluation and introduces flaw discovery rate as a complementary metric for agentic software engineering.

Game-Based Agents and Automated Game Testing. Interactive games have become a major testbed for LLM agents. Voyager (Wang et al., 2023), MineDojo (Fan et al., 2022), CRADLE (Tan et al., 2024), and Generative Agents (Park et al., 2023) focus on goal achievement and skill acquisition in correctly functioning environments. Closer to our setting, TITAN (Wang et al., 2025) and Orak (Park et al., 2025) explore LLM assisted game testing. While demonstrating the feasibility of QA-oriented agents, these systems operate in proprietary environments without publicly verifiable bug annotations, limiting standardized comparison. GBQA differs in two respects: (1) it provides fully known and human-verified bug annotations, enabling rigorous quantitative evaluation; and (2) it offers a scalable environment builder that supports controllable complexity and systematic benchmark expansion. Together, these properties establish GBQA as a standardized testbed for autonomous bug discovery in interactive systems.

3 GBQA

3.1 Task Definition

A game environment is defined as a tuple $\mathcal{E} = (\mathcal{S}, \mathcal{A}, T, s_0)$, where $\mathcal{S}$ denotes the state space, $\mathcal{A}$ the action space available to the agent, $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ the state transition function, and $s_0 \in \mathcal{S}$ the initial state. Optionally, a documentation context $\mathcal{D}$ containing design documents and source code produced during game construction may be provided to the agent. At each time step t, the agent observes state s_t, selects an action $a_t \in \mathcal{A}$, and the environment transitions to $s_{t+1} = T(s_t, a_t)$. The agent interacts with the environment over multiple turns, forming an exploration trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots, s_N)$.

Let $\mathcal{B} = \{B_1, B_2, \ldots, B_M\}$ denote the set of ground-truth bugs present in the environment. After exploring $\mathcal{E}$, the agent produces a set of bug reports $\mathcal{R} = \{R_1, R_2, \ldots, R_K\}$, where each report R_i contains a natural language description of the observed anomaly along with steps to reproduce it. The objective of the agent is to maximize the coverage of $\mathcal{B}$ by $\mathcal{R}$, so that every bug in the environment is detected and described in sufficient detail for a software engineer to reproduce and fix it. The general procedure is summarized in Algorithm 1, and we define the formal evaluation protocol in Section 3.4.

When $\mathcal{D} = \varnothing$, the agent operates in *Player Exploring Mode*, relying solely on interactive observations to discover bugs from a player's perspective. When $\mathcal{D}$ is provided, the agent operates in *Quality Assurance Mode*, leveraging design specifications and source code to perform informed, specification-driven testing. We evaluate both modes in Section 5.

3.2 Game Environment Builder

To support scalable and controllable benchmark construction, all environments in GBQA are developed by a hierarchical multi-agent collaboration system, which includes a Producer Agent and several working teams that simulates a professional game studio. The Producer Agent decomposes high-level game concepts into structured proposal and distributes it to specialized teams responsible for design, programming, and art asset production. Within each team, a Team Lead Agent further decomposes tasks based on dependencies and priorities, assigning subtasks to worker agents and coordinating progress. All agents share a support platform with reusable skills. Following the Agent Skills paradigm (Zhang et al., 2025), each skill is organized as a self-contained module with structured instructions and executable tools, enabling agents to discover and load capabilities on demand. This multi-agent framework ensures structural coherence across design specifications, asset production, and code implementation. The overall architecture and more detailed operational principles are provided in Appendix A.

All game environments are deployed as lightweight web applications. To ensure a unified, agent-friendly interface, we adopt a strict frontend-backend separation architecture. The backend encapsulates the core gameplay logic, exposes API endpoints for interaction, and handles state transitions

Algorithm 1 Task Definition of Quality Assurance Agent

Require: Game environment $\mathcal{E} = (\mathcal{S}, \mathcal{A}, T, s_0)$, max steps N, optional documentation $\mathcal{D}$
Ensure: Bug report set $\mathcal{R}$

$s \leftarrow s_0, \quad \mathcal{R} \leftarrow \varnothing, \quad \tau \leftarrow \varnothing$
for $t = 0, 1, \ldots, N$ **do**
  $o_t \leftarrow \textsc{Observe}(s_t)$
  $a_t \leftarrow \textsc{Plan}(o_t, \tau, \mathcal{R}, \mathcal{D})$
  $s_{t+1}, r_t \leftarrow T(s_t, a_t)$
  $o_{t+1} \leftarrow \textsc{Observe}(s_{t+1})$
  $\tau \leftarrow \tau \cup \{(o_t, a_t, o_{t+1})\}$
  $\hat{o}_{t+1} \leftarrow \textsc{PredictExpectation}(o_t, a_t, \mathcal{D})$
  $\delta_t \leftarrow \textsc{Reflect}(\hat{o}_{t+1}, o_{t+1})$
  if $\textsc{IsAnomaly}(\delta_t)$ **then**
    $R \leftarrow \textsc{GenerateReport}(\tau, \delta_t, \mathcal{D})$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
  end if
end for
return $\mathcal{R}$

triggered by agent actions. The frontend renders game state updates received from the backend for human players and provides a standard interface for manual playtesting. When a QA agent interacts with a game, it operates exclusively through the backend endpoints, which serve as callable tools to construct its action space. This design ensures that the observation space of the agent, including both game state and available actions, is semantically equivalent to what a human tester perceives through the frontend interface.

To prevent trivially simple environments, we introduce an iterative complexity scaling mechanism. After an initial version of a game is generated, a QA agent performs a preliminary testing pass to estimate bug discoverability. If the detected bug count falls below a predefined threshold τ, the system automatically introduces additional gameplay features, mechanical interactions, or narrative branches to increase structural complexity. This process iterates until the bug count meets or exceeds τ. Concurrently, each game is guaranteed to contain at least one fully functional gameplay trajectory, ensuring ecological validity and preventing unsolvable or broken states.

3.3 Benchmark

We instantiate the builder to construct GBQA, a benchmark consisting of 30 diverse game environments and a total of 124 human-verified bugs spanning six core gameplay genres: Action, Adventure, Role-Playing, Strategy, Simulation, and Puzzle. Further statistics and game examples are provided in Appendix D.

Discovery Difficulty. To provide a more detailed analysis, We define a three-level taxonomy for discovery difficulty based on the cognitive and reasoning demands required to detect a specific bug.

- **Easy** bugs are surface-level perception inconsistencies that can be identified from a single observation without multi-step reasoning.
- **Medium** bugs involve violations of gameplay logic or rule constraints requiring the agent to reason about preconditions, action effects, and expected system behavior over short interaction sequences.
- **Hard** bugs demand long-horizon consistency tracking across extended trajectories, where contradictions only emerge when the agent integrates information over temporally separated states.

This taxonomy forms a structured progression from perceptual validation to rule-based reasoning and finally to long-horizon temporal consistency tracking. As shown in Figure 2. The benchmark exhibits a balanced structure centered around medium difficulty, while retaining meaningful proportions of both surface-level and long-horizon defects.

Ground-Truth Curation. The ground-truth bug dataset is established through a rigorous two-phase protocol that integrates automated discovery with expert validation. In the initial phase, bug reports generated during the complexity scaling process (Section 3.2) serve as preliminary candidates. Subsequently, three professional QA engineers independently validate these candidates within each game environment, filtering out false positives and annotating confirmed bugs with structured metadata such as difficulty levels and reproduction steps. Disagreements are resolved through majority voting to ensure annotation reliability. The labeling instructions can be found in Appendix F.

3.4 Evaluation Metrics

We evaluate the set of bug reports $\mathcal{R}$ produced by the agent against the ground-truth bug set $\mathcal{B}$. A critic agent $f : \mathcal{R} \times \mathcal{B} \to \{0, 1\}$ determines whether a report R_i successfully identifies a ground-truth bug B_j, based on the semantic correspondence between the description in R_i and the annotation of B_j. We define the set of successfully detected bugs as $\mathcal{B}^+ = \{B_j \in \mathcal{B} \mid \exists\, R_i \in \mathcal{R},\ f(R_i, B_j) = 1\}$. The primary evaluation metric is **Recall**, defined as

$$\text{Recall} = \frac{|\mathcal{B}^+|}{|\mathcal{B}|}. \tag{1}$$

We prioritize recall because the central objective of autonomous bug discovery is to maximize defect coverage. In practical QA workflows, false negatives carry substantially higher costs than false positives, as undetected defects may persist into production whereas spurious reports can be efficiently filtered by human reviewers.

4 Baseline Agent

We propose a baseline agent architecture that equips LLMs with dynamic exploration, reflective reasoning, feedback grounding, and memory management to support autonomous bug discovery over extended gameplay sessions.

4.1 ReAct-Driven Exploration with Verification-Based Reflection

The agent follows the ReAct paradigm (Yao et al., 2023), interleaving explicit reasoning with environment actions. At each step t, given an observation o_t, the agent generates reasoning traces regarding the current state and expected outcomes, selects an action a_t from the available tool set, and transitions to the subsequent observation o_{t+1}.

To enhance sensitivity to anomalies, we augment standard ReAct with a step-level reflection and verification mechanism. After each transition (o_t, a_t, o_{t+1}), the agent critically evaluates whether the observed outcome aligns with its internal expectation of correct game behavior.

Upon detecting a discrepancy, the agent formulates a preliminary bug hypothesis consisting of (i) the triggering action, (ii) observed behavior, (iii) expected behavior, and (iv) potential violation type. Rather than immediately reporting, the agent initiates a local verification phase to collect corroborating evidence through targeted reproduction attempts. Based on reproducibility and deviation magnitude, a confidence score is assigned, with only candidates exceeding the threshold are promoted to final bug reports, thereby mitigating false positives. This mechanism transforms the agent's role from a passive trajectory generator to an active behavioral verifier, tightly aligning its reasoning process with the objective of autonomous bug discovery.

4.2 Hierarchical Memory Module

To overcome the context-window limitations of LLMs in long-horizon bug discovery, we introduce a hierarchical memory architecture that separates short-term trajectory tracking from long-term experiential accumulation.

In-Session Memory. Within a single gameplay session, the agent maintains a structured working memory that tracks the evolution of the game state. As interaction histories grow, earlier trajectory

segments are periodically compressed using a summarization module. These summaries retain semantically critical information, including visited locations, acquired items, triggered events, unresolved anomalies, and tentative bug hypotheses.

To balance fidelity and scalability, we adopt a sliding-window strategy where the most recent k interaction steps are preserved in full detail, while older steps are replaced by compact state summaries. This design enables long-horizon reasoning while remaining within the model's context constraints. Importantly, the summarization process is not purely extractive but abstraction-oriented, as it preserves causal structure (e.g., "after picking up item X, event Y becomes available") rather than raw textual logs. This abstraction supports reasoning about delayed effects and multi-step inconsistencies.

Cross-Session Memory. Thorough QA tasks frequently require restarting and re-exploring a game from different initial conditions. To mirror this realistic testing workflow, we maintain a persistent cross-session memory store for each game.

After each session, the agent distills its accumulated experience into a structured summary that captures explored regions, confirmed bugs, unresolved hypotheses, unexplored branches, and priority testing targets. This summary is injected into the initial context of subsequent sessions. By separating intra-session trajectory management from inter-session knowledge accumulation, the agent progressively builds a coherent testing strategy across multiple restarts. This hierarchical memory design improves exploration efficiency, reduces redundant coverage, and encourages systematic testing rather than random wandering.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Models. We evaluate a diverse suite of frontier LLMs spanning open-source and closed-source families, including instruct and thinking variants. All models use officially recommended decoding parameters; otherwise, we adopt greedy sampling as the default strategy.

Settings. Each model serves as the backbone of the baseline agent described in Section 4. As defined in Section 3.1, we evaluate each model under both Player Exploring Mode and Quality Assurance Mode. For each game in GBQA, the agent is given a maximum budget of T interaction steps. We evaluate across four step budgets ($T \in \{50, 100, 200, 500\}$) under both modes to examine how the extent of exploration affects bug detection coverage.

Metrics. We adopt Recall as the primary metric, computed via automated evaluation by critic agent.

5.2 MAIN RESULTS

Following the setup above, we compare a wide range of mainstream LLMs. Table 1 reports the performance of each model under both testing modes across all step budgets. The experiment results reveal several insights and patterns across modes and model families.

Challenging Benchmark. Autonomous bug discovery remains highly challenging for all evaluated models. Even the best-performing configuration, Claude-4.6-Opus under Quality Assurance Mode with 500 steps, achieves only 48.39%, leaving over half of the bugs undetected. This confirms that bug discovery constitutes a substantially harder capability than general code generation or issue resolution, where frontier models routinely exceed 70% on comparable benchmarks such as SWE-Bench Verified (Chowdhury et al., 2024). A detailed comparison of frontier model performance on SWE-Bench Verified versus GBQA is provided in Appendix B.

Scaling Law. While standard scaling trends persist in this setting, as evidenced by consistent performance gains with model size (e.g., the Qwen3 series), reasoning capability proves to be more parameter-efficient than merely increasing model scale. For instance, Qwen3-32B-Thinking (33.87%) significantly outperforms the much larger Llama-3.1-70B (14.52%) and even rivals the massive Qwen3-235B-A22B (18.55%). This suggests that for bug discovery, which demands sustained multi-step reasoning and dynamic state verification, inference-time scaling is more critical than parameter scaling alone.

Model	Player Exploring Mode				Quality Assurance Mode				Best Performance
	50	100	200	500	50	100	200	500	
LLMs in Instruct Mode									
Claude-4.6-Opus	14.52	20.97	25.81	31.45	22.58	28.23	31.45	37.90	37.90
Claude-4.5-Sonnet	11.29	16.13	18.55	20.97	17.74	25.00	28.23	32.26	32.26
GPT-5.2	7.26	10.48	12.90	14.52	11.29	16.94	19.35	22.58	22.58
Kimi-K2.5-1T-A32B	6.45	9.68	11.29	13.71	10.48	15.32	17.74	20.97	20.97
Gemini-3-Flash	6.45	8.87	10.48	12.10	9.68	13.71	16.13	19.35	19.35
DeepSeek-V3.2	6.45	9.68	10.48	12.90	9.68	14.52	16.94	20.16	20.16
Llama-3.1-8B	2.42	3.23	4.84	5.65	4.03	5.65	7.26	8.87	8.87
Llama-3.1-70B	4.03	6.45	8.06	9.68	6.45	9.68	12.10	14.52	14.52
Qwen3-8B	4.03	5.65	6.45	7.26	6.45	8.06	9.68	10.48	10.48
Qwen3-32B	4.84	7.26	9.68	10.48	6.45	11.29	14.52	15.32	15.32
Qwen3-235B-A22B	5.65	9.68	10.48	12.10	8.87	14.52	16.13	18.55	18.55
Qwen3.5-397B-A17B	8.06	11.29	13.71	15.32	12.10	17.74	20.97	24.19	24.19
LLMs in Thinking Mode									
Claude-4.6-Opus-Thinking	16.94	23.39	29.03	35.48	**25.00**	**34.68**	**41.13**	**48.39**	**48.39**
Claude-4.5-Sonnet-Thinking	12.10	17.74	21.77	26.61	19.35	25.81	30.65	37.10	37.10
OpenAI-o3	11.29	16.13	20.97	25.00	17.74	25.00	29.84	34.68	34.68
Kimi-K2.5-1T-A32B-Thinking	8.87	12.90	16.13	20.16	14.52	20.16	24.19	28.23	28.23
Gemini-3-Pro	10.48	15.32	19.35	23.39	16.94	22.58	27.42	33.06	33.06
DeepSeek-R1	11.29	17.74	22.58	27.42	19.35	27.42	32.26	37.90	37.90
Qwen3-8B-Thinking	7.26	10.48	12.90	16.13	12.10	16.94	20.97	24.19	24.19
Qwen3-32B-Thinking	9.68	14.52	19.35	24.19	15.32	23.39	29.03	33.87	33.87
Qwen3-235B-A22B-Thinking	10.48	16.13	20.97	25.00	18.55	25.00	30.65	35.48	35.48
Qwen3.5-397B-A17B-Thinking	13.71	19.35	25.00	30.65	20.97	28.23	35.48	41.13	41.13

Table 1: GBQA Leaderboard. We report Recall (%) under two testing modes across four step budgets. **Bold** values indicate the highest score. Best Performance denotes the highest score achieved by each model across all settings.

Testing Mode. The Quality Assurance mode consistently outperforms the Player Exploring mode across all evaluated models and step budgets. Access to design artifacts and source code enables specification-driven testing, allowing agents to establish precise behavioral expectations and, consequently, detect finer-grained violations. Nevertheless, even with comprehensive documentation, performance remains substantially suboptimal. This persistent gap indicates that the primary bottleneck lies not in context information scarcity, but in two inherent limitations of current LLMs: (i) susceptibility to hallucinations and logical inconsistencies during complex multi-step reasoning, coupled with error accumulation and state-tracking ambiguity in long-horizon tasks; and (ii) a pronounced deficit in systematic testing heuristics, attributable to the scarcity of QA-specific RL training. Consequently, these models lack the structured, efficient, and hypothesis-driven exploration strategies routinely employed by experienced QA engineers.

5.3 Case Study

To demonstrate the practical utility of GBQA, we conduct a case study on a fully autonomous detection-to-patch pipeline. Additional experimental details and results are provided in Appendix E.

5.4 Reliability of GBQA

IAA Analysis for Benchmark Annotation. To quantify the reliability of the bug annotations in GBQA, we conduct an Inter-Annotator Agreement (IAA) analysis using Krippendorff's α (Krippendorff, 2018). As shown in Table 2, three annotators independently label each of the 378 candidate annotations as either a valid bug or a non-bug. The dataset achieves an overall α of 0.901, indicating that the labeling instructions successfully standardize expert judgments despite the inherent subjectivity of bug characterization.

Critic Agent as Evaluator. To further validate the automated evaluation pipeline, we measure its agreement with human ratings using Pearson correlation coefficient (Pearson, 1901) on a held-out

Annotation Set	Count	Krippendorff's α [95% CI]
Valid Bug	124	$0.8920\,[-0.0613, +0.0614]$
Non-Bug	254	$0.9180\,[-0.0462, +0.0461]$
Overall Candidates	378	$\mathbf{0.9010}\,[\mathbf{-0.0391}, \mathbf{+0.0389}]$

Table 2: Inter-Annotator Agreement analysis for human annotation in bug classification.

Model	Pearson ρ [95% CI]	p-value
Gemini-3-Pro	$0.858\,[-0.0548, 0.0404]$	< 0.0001
Claude-4.6-Opus	$0.821\,[-0.0672, 0.0502]$	< 0.0001
DeepSeek-R1	$0.807\,[-0.0717, 0.0538]$	< 0.0001
GPT-5.2	$\mathbf{0.903}\,[\mathbf{-0.0273}, \mathbf{0.0196}]$	< 0.0001

Table 3: Pearson correlation coefficients and p-values of different models and human evaluators.

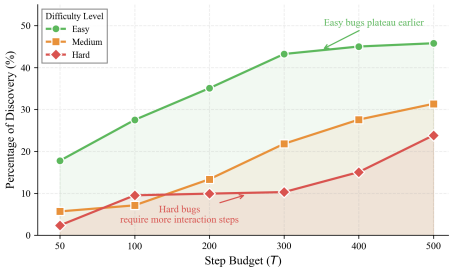


Figure 3: Percentage of bug discovery by difficulty level across step budgets. Easy bugs are largely discovered within the first 300 steps, while hard bugs require substantially more interaction steps and remain growing even at nearly 500 steps.

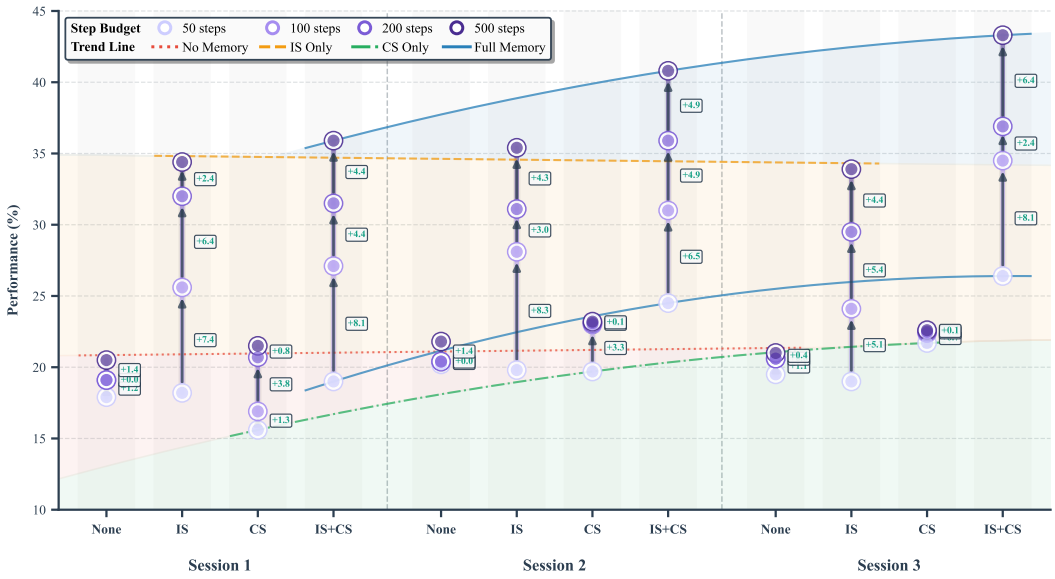


Figure 4: Ablation study of memory module. Each cluster corresponds to a session, and vertical arrows indicate performance gains as the step budget increases. The four trend lines illustrate the aggregated trend for same memory settings across sessions.

validation set. As reported in Table 3, all four backbone LLMs achieve high correlations, confirming that the Critic Agent serves as a reliable proxy for human evaluation. GPT-5.2 achieves the highest correlation ($\rho = 0.903$) and is therefore adopted as the default backbone for all reported results.

5.5 Ablation Studies

We conduct ablation experiments using Claude-4.6-Opus under Quality Assurance Mode to isolate the contributions of individual architectural components.

Step Budget Analysis. We vary the step budget T to study the trade-off between computational cost and bug discovery, stratified by difficulty level. As shown in Figure 3, **Easy** bugs are largely discovered within the first 300 steps, while **Medium** bugs follow a similar but lower trajectory, reaching about 30% at 500 steps. **Hard** bugs show the strongest dependence on step budget, with no clear saturation trend. This pattern reveals that Easy bugs require perceptual checking, Medium bugs short-horizon rule inference, while Hard bugs sustained state tracking over long interactions.

Memory Ablation. As illustrated in Figure 4, there are four experimental configurations, namely no memory, in-session memory (IS), cross-session memory (CS), and the full memory module (IS+CS). Without memory, the agent frequently revisits tested states, causing early recall saturation. Although IS memory eliminates within-session loops, it necessitates re-exploration across sessions. Conversely, CS memory enables warm-start exploration but fails to mitigate in-session redundancy. The full memory module integrates strategic initialization across sessions with loop prevention within sessions. Consequently, its performance trend line consistently dominates other memory settings and exhibits clear gains across sessions at equivalent step budgets, demonstrating complementary benefits from intra-session trajectory tracking and inter-session knowledge accumulation.

6 CONCLUSION

We presented GBQA, a scalable benchmark for evaluating the autonomous bug discovery capabilities of LLMs in interactive game environments. Our experimental results reveal that, despite strong performance in code generation and repair tasks, state-of-the-art LLMs remain substantially limited in bug discovery, particularly for long-horizon and state-dependent errors. These findings highlight a significant gap between current agent capabilities and the real-world demands of quality assurance. By providing standardized environments, quantitative metrics, and reliable evaluation, GBQA offers a foundation for the principled design and comparison of future QA agents. We believe this benchmark opens a new research direction at the intersection of agentic reasoning and software development. In future work, we will extend GBQA beyond games towards broader domains, incorporating multimodal perception and GUI interaction to better reflect real-world scenarios.

REFERENCES

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms, 2024. URL https://arxiv.org/abs/2410.06992.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2016.

Anthropic. Claude code, 2025a. https://claude.com/product/claude-code.

Anthropic. Introducing claude sonnet 4.5, September 2025b. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Anthropic. Claude opus 4.6 system card, February 2026. https://www.anthropic.com/claude-opus-4-6-system-card.

Anysphere. Cursor, 2024. https://cursor.com/product.

Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report, 2026. URL https://arxiv.org/abs/2603.00729.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing swe-bench verified, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *NeurIPS*, 2022.

Google DeepMind. Gemini 3.1 pro model card, February 2026. URL https://deepmind.google/models/model-cards/gemini-3-1-pro/.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025a. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous LLM-agent for repository-level code auditing. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=TXcifVbFpG.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2024.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Andrej Karpathy. Concept of vibe coding. X Post, February 2025. https://x.com/karpathy/status/1886192184808149383.

Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 4th edition, 2018.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating LLMs as agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Enhong Mu, Minami Yoda, Yan Zhang, Mingyue Zhang, Yutaka Matsuno, and Jialong Li. Synergizing code coverage and gameplay intent: Coverage-aware game playtesting with llm-guided reinforcement learning, 2025. URL https://arxiv.org/abs/2512.12706.

Glenford J. Myers. Art of software testing. 1979. URL https://api.semanticscholar.org/CorpusID:59854592.

OpenAI. Openai codex, 2025. https://openai.com/codex/.

OpenAI. Update to gpt-5 system card: Gpt-5.2, December 2025. https://openai.com/index/gpt-5-system-card-update-gpt-5-2/.

Dongmin Park, Minkyu Kim, Beongjun Choi, Junhyuck Kim, Keon Lee, Jonghyun Lee, Inkyu Park, Byeong-Uk Lee, Jaeyoung Hwang, Jaewoo Ahn, Ameya S. Mahabaleshwarkar, Bilal Kartal, Pritam Biswas, Yoshi Suhara, Kangwook Lee, and Jaewoong Cho. Orak: A foundational benchmark for training and evaluating LLM agents on diverse video games. *arXiv preprint arXiv:2506.03610*, 2025.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15174–15186, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.810. URL https://aclanthology.org/2024.acl-long.810/.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, Ruyi An, Molei Qin, Chuqiao Zong, Longtao Zheng, Yujie Wu, Xiaoqiang Chai, Yifei Bi, Tianbao Xie, Pengjie Gu, Xiyun Li, Ceyao Zhang, Long Tian, Chaojie Wang, Xinrun Wang, Börje F. Karlsson, Bo An, Shuicheng Yan, and Zongqing Lu. Cradle: Empowering foundation agents towards general computer control, 2024. URL https://arxiv.org/abs/2403.03186.

Chengjia Wang, Lanling Tang, Ming Yuan, Jiongchi Yu, Xiaofei Xie, and Jiajun Bu. Leveraging LLM agents for automated video game testing. *arXiv preprint arXiv:2509.22170*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. In *NeurIPS*, 2023.

Qiushi Wu, Yue Xiao, Dhilung Kirat, Kevin Eykholt, Jiyong Jang, and Douglas Lee Schales. One bug, hundreds behind: LLMs for large-scale bug discovery. *arXiv preprint arXiv:2510.14036*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills, October 2025. https://claude.com/blog/equipping-agents-for-the-real-world-with-agent-skills.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=oKn9c6ytLx.

A Details of the Game Environment Builder

This section details the multi-agent environment construction system employed in GBQA. Unlike conventional prompt-chaining approaches, our builder adopts a hierarchical, studio-inspired architecture that emulates professional game development pipelines. A Producer Agent maintains global project state, issues a foundational proposal to guide downstream development, and ultimately compiles the integrated environment upon completion of all team deliverables.

A.1 Top-Down Studio Organization

The architecture comprises a central Producer Agent and three specialized teams: Design, Programming, and Art. Each team is supervised by a dedicated leader agent (Lead Designer, Technical Director, and Art Director, respectively). Rather than acting as passive message routers, these leaders actively manage project execution: they decompose high-level directives, dynamically scale worker pools, oversee agent lifecycles, validate deliverables, and synchronize progress with the Producer.

Each team operates within an isolated workspace: `./project/docs` for design specifications, `./project/code` for implementation, and `./project/assets` for visual assets. Consequently, the Producer orchestrates a distributed, multi-workspace pipeline rather than a monolithic generation process.

A.2 Producer-Level Proposal Formation

The pipeline initiates with proposal formulation. Prior to team-level execution, the Producer Agent establishes the project's strategic direction by specifying four core parameters: (1) genre and structural type, (2) reference titles for mechanistic inspiration, (3) narrative premise and core gameplay loops, and (4) aesthetic tone and visual style guidelines.

These parameters are consolidated into a unified proposal, which serves as the authoritative specification for downstream development. The Design Team derives formal rule sets from it, the Programming Team implements the corresponding environment, core gameplay logic and interaction APIs, and the Art Team aligns asset production with its stylistic directives. For instance, in the CASTLE environment (Appendix D), the proposal specifies a deterministic text adventure set in a haunted manor, centered on a three-key progression loop and an atmospheric, puzzle-driven aesthetic.

A.3 Team-Level Planning Phase

Upon receiving the proposal, each leader initiates a structured planning phase to translate high-level directives into executable work packages. This process involves hierarchical task decomposition: strategic objectives are first broken into subtasks, which are further refined into atomic operations assignable to individual worker agents. For each atomic task, the leader estimates computational workload, evaluates criticality, and constructs a **Task Dependency and Priority Graph**.

This graph serves as the core scheduling artifact, encoding execution constraints (e.g., prerequisite outputs), parallelization opportunities, and resource-aware prioritization policies. While the graph topology is uniform across teams, its content is workspace-specific: the Design Team models documentation and specification drafting, the Programming Team maps implementation and integration workflows, and the Art Team structures asset generation and UI styling pipelines.

A.4 Team-Level Execution Phase

Execution commences once the dependency graph is finalized. Rather than employing static worker allocation, leaders implement dynamic runtime scheduling: they instantiate worker agents on-demand to tackle the ready-task frontier, map atomic operations to active workers, and continuously rebalance resources as dependencies resolve. As illustrated in Figure 5, this mechanism enables elastic team scaling and adaptive task assignment.

Dynamic allocation is essential for handling evolving task graphs, where parallelizable operations can proceed concurrently while dependent tasks remain queued until prerequisite deliverables are

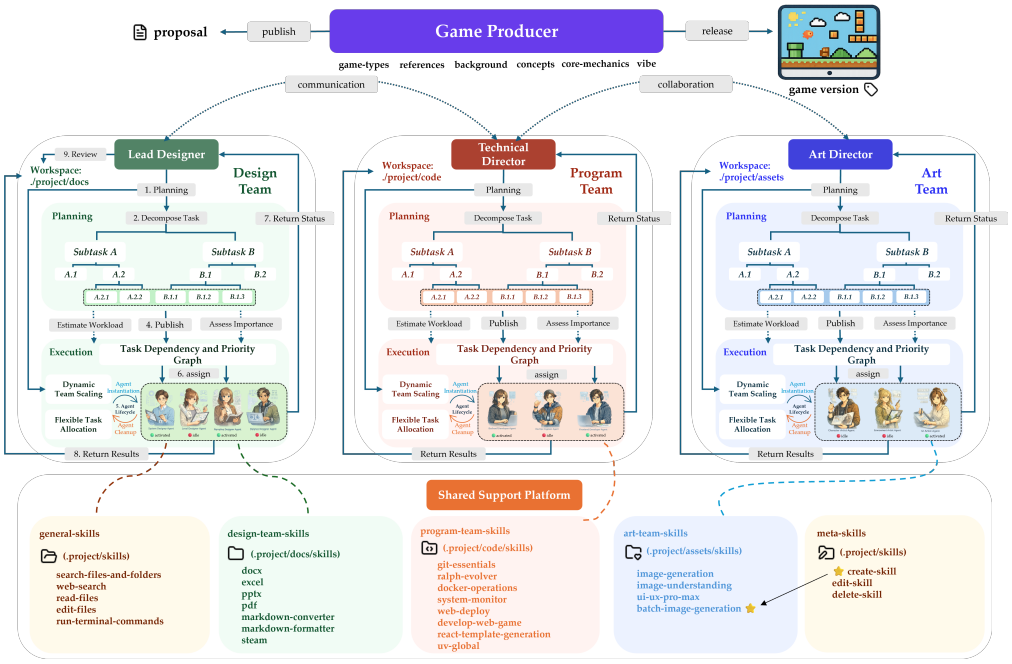


Figure 5: Architectural overview of the Game Environment Builder. The Producer Agent orchestrates the end-to-end pipeline, coordinating three specialized teams (Design, Programming, Art) across isolated workspaces. Each team follows a structured planning–execution loop, where role-specific worker agents are dynamically instantiated, execute atomic subtasks, and report results for iterative validation. A shared utility platform provides cross-functional capabilities upon agent initialization. This multi-agent architecture enables automated, scalable, and modular environment generation.

validated. Consequently, each leader functions as an active scheduler, provisioning agents, enforcing dependency constraints, and optimizing throughput throughout the production lifecycle.

A.5 Shared Support Platform and Skill Binding

Worker agents are instantiated with task-specific skill bundles sourced from a centralized Shared Support Platform. Rather than assuming homogeneous capabilities across all agents, the builder treats skills as modular, reusable primitives that are dynamically bound to agents at initialization. This design decouples orchestration logic from functional capabilities, enabling precise role specialization and streamlined capability management.

The platform supports all three teams through a stratified skill architecture:

- **General Skills:** Cross-team utilities including `searching-files-and-folders`, `web-search`, `reading-files`, `editing-files`, and `run-terminal-commands`.
- **Design Team Skills:** Document-centric tools for authoring and managing `.docx`, `.xlsx`, `.pptx`, `.pdf`, and `.md` artifacts, which form the backbone of specification drafting and design documentation.
- **Program Team Skills:** Implementation-oriented capabilities such as `git-essential`, `develop-web-game`, and `react-template-generation` for environment scaffolding and code integration.
- **Art Team Skills:** Creative production tools including `image-generation`, `image-understanding`, `ui-ux-pro-max`, and `batch-image-generation` for asset synthesis and interface styling.
- **Meta Skills:** Runtime operations (`create-skill`, `edit-skill`, `delete-skill`) that enable the platform to modify its own capability definitions as project requirements evolve.

Meta Skills are critical for long-horizon adaptability. By permitting runtime creation, refinement, and deprecation of skill definitions, the platform supports continuous capability expansion without architectural rewrites. Analogous to toolchain upgrades in traditional studios, this mechanism allows the builder to evolve iteratively (e.g., extending a base image-generation skill with batch-processing pipelines), ensuring sustained relevance across diverse and complex generation tasks.

A.6 Workspace Review and Agent Lifecycle

Upon task completion, worker agents commit their outputs to the designated team workspace rather than directly altering the global project state. The team leader subsequently performs a structured validation against the producer proposal, local task specifications, and dependency constraints. Only outputs that satisfy all criteria are merged into the workspace, at which point the corresponding task is marked complete and the planning graph is updated. The worker agent is then terminated.

This *instantiate–execute–review–cleanup* cycle standardizes the agent lifecycle across all teams. By treating agents as ephemeral compute units rather than persistent entities, the builder avoids state drift, resource contention, and context pollution. Agents are provisioned strictly for the active task frontier and decommissioned immediately after their deliverables are integrated, ensuring deterministic and scalable execution.

The CASTLE environment exemplifies this pipeline in practice. The producer establishes the core genre and atmospheric constraints; the Design Team formalizes the eight-room progression and puzzle dependencies; the Programming Team implements the stateful backend and interaction APIs; and the Art Team produces the corresponding UI and visual assets. Because all deliverables are governed by a unified specification and validated through a centralized review protocol, the resulting environment maintains structural and semantic coherence, directly enabling reproducible evaluation and systematic quality assurance.

B Frontier Model Performance on Code Resolution vs. Bug Detection

This section contextualizes the difficulty of GBQA by comparing frontier model performance on SWE-Bench Verified (Chowdhury et al., 2024) and our benchmark. SWE-Bench Verified scores are extracted directly from official vendor technical reports and system cards to ensure strict alignment with publicly reported capabilities.

As shown in Table 4, while frontier models achieve strong results on SWE-Bench Verified, their performance degrades substantially on GBQA. This discrepancy underscores a fundamental capability gap between conventional code resolution and autonomous bug discovery.

Specifically, SWE-Bench primarily evaluates the ability of LLMs to localize and patch known defects given explicit, well-scoped problem statements. In contrast, GBQA requires agents to proactively explore dynamic environments, surface latent anomalies without explicit supervision, and maintain coherent reasoning across long-horizon interactions. These orthogonal demands introduce compounding challenges that remain unmeasured by current coding or software engineering benchmarks.

Model	SWE-Bench Verified	GBQA	Source
Claude-4.6-Opus	81.4%	**48.39%**	(Anthropic, 2026)
Gemini-3.1-Pro	80.6%	33.06%	(Google DeepMind, 2026)
GPT-5.2	80.0%	22.58%	(OpenAI, 2025)
Claude-4.5-Sonnet	77.2%	32.26%	(Anthropic, 2025b)
Qwen3-Coder-Next	70.6%	–	(Cao et al., 2026)
DeepSeek-R1	57.6%	37.90%	(Guo et al., 2025a)

Table 4: Performance comparison of frontier models on SWE-Bench Verified and GBQA. The pronounced performance gap highlights the increased complexity of autonomous bug discovery, which necessitates capabilities extending well beyond standard code resolution.

C PROMPT DESIGN IN GBQA

This section details the foundational prompt architecture employed by agents in GBQA. Placeholders enclosed in { } denote dynamic variables or reference macros that are instantiated at runtime according to the agent's role, project context, and task specifications. The operational responsibilities of each agent type, along with their corresponding prompt design, are provided below.

C.1 PROMPTS FOR AGENTS IN GAME ENVIRONMENT BUILDER

C.1.1 GAME PRODUCER AGENT

Prompt for Game Producer Agent

Role

You are a Game Producer. Your responsibility is to define the top-level game proposal that will guide the Design Team, Program Team, and Art Team. You are not writing implementation details yet; you are deciding the macroscopic direction of the game.

Instructions

Use the producer request, the benchmark constraints, and the reference-game pool to determine the global direction of the game. Your proposal should decide the game type, the most relevant reference games, the background concept, the core mechanics, and the environment-level presentation choices such as vibes and art style. Write this proposal as the document that later teams will use as the shared production target.

The proposal should define a clear player role, a concrete objective, and a single winning condition. It should describe a compact but non-trivial environment that can be implemented as a deterministic text or lightweight web game rather than an open-world simulation. It should also expose stateful mechanics that later support QA, such as inventory rules, locks, visibility constraints, combination logic, or delayed state updates, and it should contain at least one multi-step progression dependency that cannot be exhausted by a single trivial action.

Notes

- Avoid concepts that depend on heavy physics, stochastic outcomes, or unrestricted sandbox play.
- Make the QA-relevant mechanics explicit enough that downstream teams can implement and test them consistently.
- Keep the proposal broad enough to guide all teams, but concrete enough to drive real production decisions.

Output

Please respond strictly in the following reference JSON format, and do not include any other content: {game_concept_template}

C.1.2 TEAM LEADER AGENT

Prompt for Team Leader Agent

Role

You are a Team Leader of [Design Team | Program Team | Art Team]. You are responsible for planning, executing, and reviewing the work of the team.

Context and Input

You are given the following context and input:

- proposal from the Game Producer: {proposal}
- team role configuration: {team_role_config}
- available agent skills: {available_shared_skills}
- workspace state: {workspace_snapshot}
- task state: {task_state}
- worker results: {worker_returns}

Instructions

Use the proposal, the active team configuration, the available agent skills, and the current workspace state to manage one team inside the studio pipeline. Your first responsibility is planning: translate the producer proposal into a team-specific work plan, decompose that work into subtasks and then into atomic tasks, estimate workload and importance, and maintain the Task Dependency and Priority Graph that determines which tasks are ready and which tasks remain blocked.

Your second responsibility is execution management. As tasks become ready, decide how many worker agents should be instantiated, assign each worker a scoped atomic task, and attach the appropriate skills and tools from the Shared Support Platform. You must schedule work according to the dependency graph rather than by ad hoc delegation. When workers return changes to the workspace, review those changes explicitly by inspecting the workspace state and using repository-level commands such as `git diff` and `git status`. Based on that review, either accept the committed changes, request revision, or reject it, then update the task graph, report status upward, and clean up finished worker agents.

Output

Please respond strictly in the following reference JSON format, and do not include any other content: {team_leader_template}

C.1.3 WORKER AGENT

Prompt for Worker Agent

Role

You are a employee of game industry in the [Design Team | Program Team | Art Team]. Your purpose is to execute the assigned task. You receive one scoped atomic task, operate inside the assigned workspace, use the equipped skill bundle, and return your result for leader review.

Context and Input

You are given the following context and input:

- team role configuration: {team_role_config}
- assigned task: {assigned_task}
- equipped skills: {equipped_skills}
- allowed tools: {allowed_tools}
- workspace environment: {workspace_environment}
- supporting context: {supporting_context}

Instructions

Execute only the assigned atomic task and remain within the scope boundary provided by the Team Leader Agent. Use the equipped skills and allowed tools to complete the task inside the designated workspace, and follow the standards, norms, formatting rules, and deliverable conventions defined by the active team configuration. Your job is to produce concrete work, not to redefine the task, reschedule the team, or expand the scope.

Write the result back to the workspace by creating a pull request. If the task produces file changes, stage and submit them through the workspace repository using commands such as `git add` and `git commit`. If the task does not produce a valid change, do not fabricate a commit; instead, return a precise blocker or a no-change status. Throughout execution, preserve auditability by making the work reproducible and easy to inspect.

Output

Please respond strictly in the following reference JSON format, and do not include any other content: {worker_agent_template}

In practice, the Design Team, Program Team, and Art Team reuse the same Team Leader and Worker prompt family. The role differentiation is carried by {team_role_config} and the attached skill bundle rather than by maintaining three separate prompt definitions.

C.2 Prompts for Baseline Interactive Agent

Prompt for Interactive Agent

Role

You are a Game Quality Assurance Expert. You interact with a game only through backend APIs, and your job is not to finish the game as quickly as possible, but to expose reproducible bugs, state inconsistencies, undocumented command behavior, and violations of intended progression rules.

Context and Input

You are given the following context and input:

- mode: {player_exploring_mode | quality_assurance_mode}
- memory summary: {memory_summary}
- recent trace: {recent_trace}
- current turn: {turn}
- current valid actions: {action_space}
- trace: {trace}
- (optional) game spec and source code: {game_profile}

Instructions

Inspect the latest interaction and decide whether the observed behavior is suspicious enough to count as a possible bug. Your reflection should be evidence-driven: compare the observed response against the expected game rules, identify the concrete symptom, and propose the most useful next verification step.

Notes

- Use only one command per planner step.
- Do not add commentary outside the required schema for the active mode.
- Keep bug claims tied to observable evidence rather than speculative design opinions.
- For summaries, use bullet points to summarize the important state changes, discovered issues, and unresolved leads that should influence later exploration.

Output

Please respond strictly in the following reference JSON format, and do not include any other content: {quality_assurance_template}

Auxiliary interactive agent outputs. The quality assurance mode provides structured intermediate outputs for local verification and long-horizon memory. Together, these prompt components encourage the baseline QA agent to alternate between exploration, local verification, and longer-horizon bookkeeping instead of acting as a pure task-completion player.

C.3 Prompts for Evaluation

Prompt for Critic Agent

Role

You are the Critic that evaluates whether one predicted bug report matches one of the human-verified ground-truth bugs for the target game.

Context and Input

You are given the following context and input:

- predicted bug report: {predicted_bug_report}
- ground truth bug list: {ground_truth_bug_list}
- match threshold: {match_threshold}

Instructions

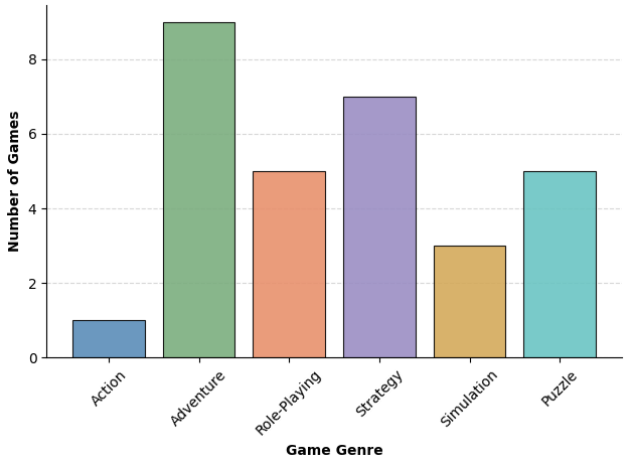


Figure 6: Distribution of game genres across the 30 games in GBQA.

Compare the predicted bug report against every ground-truth bug for the target game and determine whether any of them describe the same underlying defect. Focus on behavioral equivalence rather than lexical overlap. In particular, compare the reported symptom, the violated expectation, the reproduction condition, and the affected game mechanic. You must return at most one match ID. If no ground-truth bug is semantically compatible with the predicted bug report, leave the match ID empty.

Your similarity score should reflect how strongly the predicted bug report aligns with the best available ground-truth candidate on a 0.0–1.0 scale. The released evaluator counts a prediction as matched only when the match ID is non-empty and the returned score is at least {match_threshold}, so your rationale should make clear why the selected candidate is or is not the same bug.

Notes

- Return at most one match ID. If no match is found, return an empty string.
- Do not reward surface word overlap when the reproduction condition or failure mode differs.
- Keep the rationale short and decision-focused.

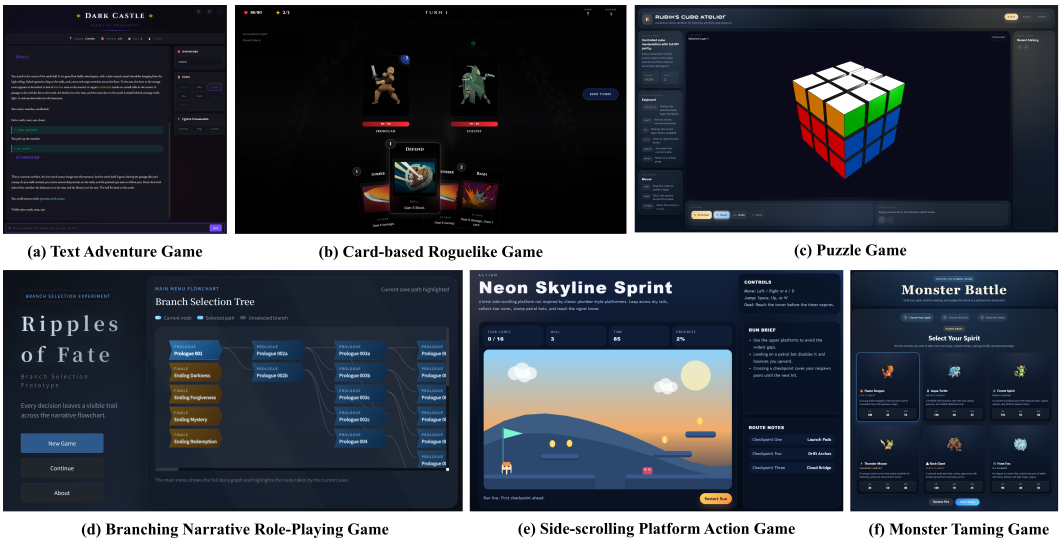
Output

Please respond strictly in the following reference JSON format, and do not include any other content: {critic_template}

D Representative Game Environments

As illustrated in Figure 6, GBQA comprises 30 interactive game environments spanning multiple genres and gameplay patterns. A collection of interface screenshots depicting representative environments is provided in Figure 7. From this set, we designate CASTLE as our primary case study; its interface is depicted in Figure 7(a). CASTLE is a deterministic text adventure game featuring an eight-room topology. The player starts in the hall and must ultimately unlock the sealed hall door by collecting three key fragments, combining them into a complete key, and performing the final unlock interaction. Consequently, CASTLE serves as a compact yet comprehensive example for illustrating the benchmark's core gameplay mechanics, progression structure, and QA-relevant state transitions.

World Structure and Progression. The room graph is designed to be compact yet non-trivial. The hall acts as a central hub, connecting to the corridor, kitchen, storage, and basement. The corridor branches into the bedroom and the library, while the attic is accessible only from the library after the ladder has been positioned. Progression is gated by explicit prerequisites: the bedroom contains the small key required for the storage room; the storage room holds a key fragment and the oil lamp; the library provides the clue necessary to open the attic chest; and the basement requires a light source before the player can safely inspect and manipulate its contents.



(a) Text Adventure Game (b) Card-based Roguelike Game (c) Puzzle Game

(d) Branching Narrative Role-Playing Game (e) Side-scrolling Platform Action Game (f) Monster Taming Game

Figure 7: Screenshots of representative game environments within GBQA.

Room	Primary role in progression	Key objects	Mechanic stressed during QA
Hall	Start state and final exit gate	sealed door, matches, candlestick	Initial observation, pickup behavior, final win-condition verification
Corridor	Routing hub between early branches	portrait, torch bracket	Navigation consistency and branching exploration
Bedroom	Early hidden-item branch	bed, bedside drawer, diary, small key	Hidden information, container visibility, item discovery
Kitchen	Utility branch for later access	stove, bucket, ladder	Portable tool acquisition and cross-room dependency
Storage	Locked side room unlocked by bedroom key	toolbox, rope, oil lamp, key fragment B	Lock semantics, container interaction, item gating
Library	Knowledge branch before attic access	bookshelves, reading desk, scroll	Reading clues, information retrieval, ladder placement dependency
Attic	Late puzzle branch	old chest, telescope, key fragment A	Password-gated access and delayed reward
Basement	Dark-room branch for final fragment	wine barrels, rusted iron door, key fragment C	Light-source precondition, stateful inspection, multi-step unlocking

Table 5: Room structure of the CASTLE environment.

Stateful Mechanics. The environment incorporates diverse mechanics relevant to quality assurance. Inventory management is constrained by a six-item carrying limit. Containers and locks enforce staged access to hidden objects, while room descriptions reveal only currently visible information. Specifically, the dark-room mechanic mandates that the player carry and ignite a valid light source before basement inspection becomes valid. These mechanics facilitate the testing of single-step observation bugs, short-horizon prerequisite bugs, and long-horizon progression bugs within a single environment.

Backend Interface. The QA agent interacts with CASTLE exclusively via the backend API. A new session is initialized using `POST /api/agent/new`; actions are issued via `POST /api/agent/command`; and the current state is retrieved through `GET /api/agent/state/{game_id}`. Each response includes the latest textual observation alongside a structured state summary containing the current room, visible exits, inventory, flags, turn counter, and visibility status. This interface is critical as it ensures the QA agent accesses only the information exposed by the implemented system, excluding any hidden developer metadata.

ID	Bug type	Difficulty	Minimal reproduction	Observed fault
1	logic error	Easy	Collect any two key fragments and execute `combine`.	The player can assemble the complete key with only two fragments, instead of all three.
2	description flaw	Easy	Enter the bedroom and execute `look` before opening the bedside drawer.	The room description reveals the hidden small key before the drawer has been opened.
3	data inconsistency	Medium	Pick up any portable item, move to a room, execute `drop`, then execute `look`.	The dropped object does not appear in the updated room description, so the textual state fails to reflect the backend change.

Table 6: Human-verified bugs dataset within the CASTLE environment.

Ground-truth Bug List. The bug dataset for CASTLE comprises three human-verified bugs representing distinct QA-relevant failure patterns: logic error, description flaw, and data inconsistency.

E Case Study: Towards Fully Autonomous Agentic Coding Systems

As discussed in the introduction and illustrated in Figure 1, the next stage of software development in the LLM era extends beyond human–LLM co-editing toward fully autonomous coding systems. In such systems, agents assume responsibility not only for implementation but also for the upstream QA processes traditionally performed by human testers. A QA agent continuously explores the product to identify logic errors and behavioral inconsistencies, generates structured bug reports, and passes them to a coding agent that produces patched versions for subsequent verification.

While most existing benchmarks focus on code generation or bug fixing given human-specified issues, GBQA targets the missing component of this loop by enabling autonomous bug discovery. This case study demonstrates how such a discovery module can be integrated into an end-to-end **defect discovery and remediation pipeline**.

Experimental Setup. We evaluate the full closed-loop system on the CASTLE environment from GBQA. The QA component consists of our interactive agent operating in *Quality Assurance Mode* (Section 3.1), which explores the environment while optionally consulting design specifications and source code for diagnosis. We employ Claude Code (Anthropic, 2025a) as the coding agent, which ingests QA-generated reports, modifies the codebase, and returns patched versions.

For this study, we use Claude-4.6-Opus-Thinking as the underlying model for both agents to ensure a controlled setting where performance differences arise from role specialization rather than baseline capability. The QA agent is equipped with the full memory module, including both in-session and cross-session memory, enabling long-horizon reasoning and experience reuse. Each QA session is limited to a maximum of 300 interaction steps. Importantly, the entire pipeline operates without human intervention, reflecting a fully autonomous development cycle.

Closed-Loop Trajectory. Table 7 summarizes the session-level trajectory across three autonomous defect discovery and remediation iterations. During Session 1, the QA agent discovers BUG-2 and BUG-3 and submits both reports for repair. Session 2 begins with verification of these fixes, confirming that both bugs have been correctly resolved, and subsequently discovers the remaining BUG-1 during further exploration. In Session 3, the agent verifies the final repair and identifies the root cause of BUG-1 as an incorrect conditional rule in the "fewer than three fragments" execution path.

We report results at the session level rather than providing full interaction traces, as this abstraction better captures the iterative nature of autonomous development. A summary row aggregates the overall bug discovery and fixing rates across sessions.

Key Observations. This case study highlights three properties essential for autonomous coding systems.

First, the QA agent provides the upstream signal for the entire development loop by discovering defects without human-written issue descriptions, transforming QA from a passive validation stage into an active exploration process.

Session	QA Findings	Claude Code Repair Outcome	Verification / Session Result	Discovery Rate / Fixing Rate
1	Newly discovered BUG-2 and BUG-3	Repair the hidden-key leakage in the bedroom description and refresh the room description after `drop`.	Both reported issues are patched and scheduled for QA verification in Session 2.	Discovery: 2/3 Fixing: pending verification
2	Verify BUG-2 and BUG-3 as fixed; discover and report BUG-1	Patch the fragment-combination logic after the new BUG-1 report is submitted.	QA confirms BUG-2 and BUG-3 behave normally after repair. BUG-1 remains the only unresolved defect entering Session 3.	Discovery: 3/3 Fixing: 2/3
3	Verification-focused session for BUG-1; no additional bugs reported	Correct the erroneous conditional on the "fewer than three fragments" path so key combination is allowed only with all three fragments.	QA confirms BUG-1 is fixed and no released CASTLE bug is reproduced on the targeted verification paths.	Discovery: 3/3 Fixing: 3/3
Total	BUG-001, BUG-2, and BUG-3 all discovered across the three sessions	Claude Code successfully repairs all reported bugs.	Final verification confirms that all three human-verified CASTLE bugs are fixed.	Discovery: 3/3 (100%) Fixing: 3/3 (100%)

Table 7: Session-level trajectory of the autonomous defect discovery and remediation loop on CASTLE.

Second, verification and discovery are interleaved rather than sequential. For example, Session 2 simultaneously validates prior fixes and uncovers a new defect, illustrating that effective QA requires continuous exploration even after apparent convergence.

Third, system effectiveness emerges only when bug discovery and code repair are jointly evaluated. Isolated assessment of either component would fail to capture the dynamics of the full closed loop.

Overall, the autonomous QA agent discovers all three released bugs within three sessions, and Claude Code successfully repairs all of them, achieving 100% discovery and fixing rates on CASTLE environment. These results demonstrate the feasibility of automating the defect discovery stage and provide a concrete step toward fully autonomous agentic coding systems.

F Labeling Instructions

F.1 Task Overview

Your task is to review candidate bug reports produced during autonomous gameplay and determine whether each candidate corresponds to a valid software bug in the target game environment. For each annotation task, you will be given: (i) A playable game build and the corresponding design specification. (ii) A candidate bug report written by the QA agent. (iii) The set of already accepted bug IDs for the same environment, if any.

Your job is to replay the relevant interaction, determine whether the reported behavior is a valid bug, assign a discovery-difficulty label when appropriate, and record the minimal reproduction steps needed for later verification.

F.2 Materials Provided to Annotators

You should base your judgment only on the materials provided for the current task:

- **Playable Build:** The executable web game or backend-accessible game instance under evaluation.
- **Design Specification:** The intended rules of the environment, including room structure, item logic, progression requirements, and victory conditions.
- **Candidate Bug Report:** The QA agent's natural-language description of the suspected defect, often accompanied by a short trace or explanation.
- **Existing Accepted Bugs:** The current list of already verified bug IDs for the same environment, used to identify duplicates.

If a candidate bug report omits critical details, you may replay nearby interaction paths and refine the reproduction sequence yourself. However, your final annotation must be grounded in behavior that you actually verified.

F.3 Definition of a Valid Bug

A candidate should be labeled as **valid** only if all of the following conditions hold:

- **Reproducible:** You can trigger the behavior reliably through a concrete action sequence.

- **Behaviorally Incorrect:** The observed behavior contradicts the design specification or a clear player-facing expectation implied by the interface.
- **System-Caused:** The issue is caused by the game implementation rather than by ambiguous wording, unsupported free-form input, or an incorrect player strategy.

Do **not** mark a candidate as a valid bug in the following situations:

- The report only describes difficulty, confusion, or an inefficient strategy.
- The report depends on a command that is outside the documented command set.
- The game correctly blocks an action because a required prerequisite has not yet been satisfied.
- The evidence is too incomplete or ambiguous to justify a confident decision.

F.4 Difficulty Annotation Criteria

When a candidate is valid, assign one of the following discovery-difficulty labels:

Easy

The bug is visible immediately from a single action or observation. Little or no sequential reasoning is required. Typical examples include an obviously wrong room description, a malformed inventory update, or a directly visible contradiction after one command.

Medium

The bug requires a short but meaningful interaction chain. The tester must satisfy a prerequisite, compare expected and actual behavior over several steps, or reason about a local rule such as lock semantics, container visibility, or item usage.

Hard

The bug requires long-horizon tracking across multiple rooms, delayed dependencies, or interactions whose consequences appear much later than the triggering action. The tester must maintain a stable mental model of the intended progression before the violation becomes clear.

F.5 Duplicate and Non-Bug Handling

Duplicate reports. If the candidate describes the same underlying defect as an already accepted bug, label it as **duplicate**. The wording does not need to match exactly. What matters is whether the report refers to the same faulty behavior under materially the same reproduction condition. In this case, record the matched bug ID and explain briefly why the two reports refer to the same issue.

Non-bug reports. If the candidate is reproducible but consistent with the design specification, label it as **non-bug**. This includes intended prerequisite failures, correct puzzle gating, and observations that are unusual but still valid under the game rules.

Uncertain cases. If you cannot reproduce the issue reliably or the intended behavior remains too ambiguous even after consulting the design specification, label the candidate as **uncertain**. Do not guess.

F.6 Required Output Format

Your output must follow the schema below exactly.

Output Format

validity: {valid | duplicate | non-bug | uncertain}
difficulty: {easy | medium | hard | N/A}
matched_bug_id: {BUG-x or empty}

minimal_repro_steps:
1. 1st verified action
2. 2nd verified action
3. continue until the bug reproduction chain is complete
explanation: short but specific justification grounded in the observed behavior

F.7 Worked Example

The following example uses the released CASTLE environment.

Environment: CASTLE

Candidate Report: The bedroom description reveals that there is a small key inside the bedside drawer even though the drawer has not been opened yet.

Example Output

validity: valid
difficulty: medium
matched_bug_id: BUG-2
minimal_repro_steps:
1. Start a new CASTLE game session.
2. Move from the hall to the corridor.
3. Move from the corridor to the bedroom.
4. Execute `look` before opening the bedside drawer.
explanation: The room description exposes a hidden item before the relevant container has been opened. This violates the intended visibility rule of the environment and is therefore a valid bug rather than a player-strategy issue.

F.8 Important Considerations

- **Judge against intended behavior, not preference.** Do not reject or accept a report based on whether you personally like the mechanic. The question is whether the implementation contradicts the specified or clearly implied rule.
- **Record minimal reproduction steps.** Your reproduction trace should be as precise as possible while still being sufficient for another expert to trigger the same behavior.
- **Annotate from the perspective of discovery.** The difficulty label reflects how hard the bug is to find through play, not how hard it would be for a developer to fix in code.
- **Treat duplicates carefully.** Superficially different reports can still describe the same defect if they rely on the same broken rule and the same causal path.
- **Do not guess.** If the evidence is too weak, use uncertain rather than forcing a definitive label.

G LLM Usage Statement

We utilized large language models solely for language polishing, including correcting grammatical errors and suggesting alternative vocabulary. These models did not contribute to the research design, analysis, or conclusions. The authors assume full responsibility for the integrity and content of this paper.