SAGE: Semantic-Aware Gray-Box Game Regression Testing with Large Language Models

Jinyu Cai[1], Jialong Li[1], Nianyu Li[2], Zhenyu Mao[3], Mingyue Zhang[4*], Kenji Tei[5]

[1]Waseda University, Tokyo, Japan.
[2]Independent Researcher, Beijing, China.
[3]City University of Hong Kong, Hong Kong, China.
[4]Southwest University, *Chongqing, China.
[5]Institute of Science Tokyo, Tokyo, Japan.

*Corresponding author(s). E-mail(s): myzhangswu@swu.edu.cn;
Contributing authors: bluelink@toki.waseda.jp; lijialong@fuji.waseda.jp;
li_nianyu@pku.edu.cn; zhenyumao2-c@my.cityu.edu.hk; tei@comp.isct.ac.jp;

Abstract

The rapid iteration cycles of modern live-service games make regression testing indispensable for maintaining quality and stability. However, existing regression testing approaches face critical limitations, especially in common gray-box settings where full source code access is unavailable: they heavily rely on manual effort for test case construction, struggle to maintain growing suites plagued by redundancy, and lack efficient mechanisms for prioritizing relevant tests. These challenges result in excessive testing costs, limited automation, and insufficient bug detection. To address these issues, we propose SAGE, a semantic-aware regression testing framework for gray-box game environments. SAGE systematically addresses the core challenges of test generation, maintenance, and selection. It employs LLM-guided reinforcement learning for efficient, goal-oriented exploration to automatically generate a diverse foundational test suite. Subsequently, it applies a semantic-based multi-objective optimization to refine this suite into a compact, high-value subset by balancing cost, coverage, and rarity. Finally, it leverages LLM-based semantic analysis of update logs to prioritize test cases most relevant to version changes, enabling efficient adaptation across iterations. We evaluate SAGE on two representative environments, Overcooked Plus and Minecraft, comparing against both automated baselines and human-recorded test cases. Across all environments, SAGE achieves superior bug detection with significantly lower execution cost, while demonstrating strong adaptability to version updates.

Keywords: Game Playtesting, Regression Testing, Gray-box Testing, Large Language Models

1 Introduction

In recent years, digital games have evolved into large-scale, continuously updated software systems. Under the "game-as-a-service" model, developers frequently release content updates—such as new quests, gameplay elements, and balance adjustments—to continuously evolve the game system [1] However, each update inevitably modifies part of the game logic or data, introducing potential inconsistencies or regressions that may compromise existing functionalities, degrade user experience, or cause economic loss [2]. Because modern games typically involve complex state spaces, real-time interactions, and tightly coupled subsystems, ensuring the functional stability of updates within a fast-paced iterative workflow has become a major technical challenge in game development.

Within this context, regression testing is widely regarded as a key technique to ensure game quality. By re-executing existing test cases, regression testing verifies whether new versions preserve previously implemented functionalities [3]. However, given the enormous state spaces and high uncertainty of game environments, existing regression testing methods are still highly dependent on manual execution, consuming both time and human resources. It is reported that regression testing can account for more than 80% of the total testing cost in real projects [4]. As a result, many companies only conduct full-scale regression testing during major version releases, leaving potential quality risks in frequent minor updates. Consequently, the development of efficient and automated game regression testing methods has become an urgent research direction.

Early regression testing relied heavily on human testers to manually execute and verify tasks. While this approach ensured reliability, it imposed prohibitive labor costs. To alleviate this issue, researchers proposed record-and-replay mechanisms [5], which capture real player operations and transform them into reusable test cases, thereby reducing repetitive effort. Later, [6] developed a GUI-based recording framework that simplified test case construction, enabling testers without programming expertise to produce high-quality scripts. With the advent of machine learning, reinforcement learning (RL) methods were introduced into game testing [7]. Unlike fixed action sequences, RL agents learn behavioral strategies by dynamically interacting with the environment, with the learned policy itself serving as a test case. This allows RL agents to actively explore and cover a broader set of potential scenarios. More recently, GameRTS [8] advanced the systematization and efficiency of regression testing by introducing a white-box regression test selection (RTS) method. By analyzing source code and resources, their approach can intelligently identify the most relevant test cases to re-execute, effectively avoiding unnecessary redundancy while maintaining a high bug detection rate.

However, testing teams, especially in large game development teams, are often independent or outsourced from the development team, and operate in gray-box settings. This setting represents a realistic condition: while the source code and internal implementation details are inaccessible (unlike white-box testing), testers can still access and observe structured runtime information, such as game APIs, event callbacks, runtime logs (e.g., object states, player actions), and high-level developer documentation (e.g., changelogs). Under this practical constraint of the gray-box setting, how to achieve efficient and automated regression testing methods remains a challenge. Specifically, this can be subdivided into the following three problems: (i) the Foundation Issue (test suite construction): Regression testing often presupposes

the existence of a high-quality, high-coverage test suite. However, in complex gray-box game projects, constructing this initial, high-quality suite is itself a significant challenge. Specifically, the existing RL-based exploration suffers from low efficiency and local optima, while traditional record-and-replay methods depend heavily on human effort and lack scalability. (ii) the Maintenance Issue (test suite maintenance): As game versions are continuously updated, the foundational test suite continuously grows in size, and it becomes progressively filled with outdated, invalid, and redundant test cases. Therefore, the second challenge is how to maintain a compact, high-value static test suite that maximizes regression detection while minimizing wasteful execution. (iii) the Selection Issue (test suite selection): Given a high-quality, well-maintained general-purpose suite, the rapid update frequency often makes it unrealistic to execute the entire suite within limited time constraints. Therefore, the third challenge is: given a specific version update and its associated documentation, how can we dynamically identify and prioritize the subset of test cases from the test suite that are most relevant to the changes, to maximize the bug-detection rate within a constrained testing window.

While gray-box testing restricts direct access to source code, it exposes rich, human-understandable semantic artifacts—runtime logs, event traces, and changelogs—that reflect a game's underlying logic and behavioral dynamics. Modern large language models (LLMs) excel at semantic understanding and reasoning; they can interpret these artifacts and ground them into machine-executable actions, enabling systematic regression testing without internal code analysis. We therefore propose SAGE, a semantic-aware regression testing framework for gray-box game environments. At its core, SAGE installs the LLM as a semantic orchestrator, transforming heuristic practices into a systematic, data-driven process that operates entirely on log-level information. To realize this vision, SAGE systematically resolves the three aforementioned issue through three interconnected mechanisms: (1) for the Foundation Issue, a semantic-driven generation mechanism where an LLM guides a reinforcement learning (RL) agent to conduct efficient, goal-oriented exploration and automatically assemble a diverse foundational test suite; (2) for the Maintenance Issue, a semantics-informed multi-objective optimization process that balances cost, coverage, and behavioral rarity, refining the expanding repository into a compact, high-value static subset; (3) for the Selection Issue, an LLM-based update interpreter that reads natural-language change logs in the gray-box setting and translates version deltas into test priorities, dynamically directing resources toward the most relevant test cases.

Our contributions are as follows:

- We design SAGE, an end-to-end unified gray-box regression testing framework. It automates test case generation, multi-objective optimization, and update-aware test case prioritization into a closed-loop, semantic-driven process, systematically addressing the core challenges in gray-box game testing scenarios.
- We propose a test case optimization and selection method that uses semantics as a bridge. The core of this method lies in its use of LLMs to translate high-level, unstructured information (e.g., abstract test goals, natural language update logs) into structured, machine-executable testing strategies (e.g., quantifiable optimization metrics, precise test priorities), enabling intelligent test decisions in environments without source code.

- We conduct a comprehensive empirical study in two complex game environments—Overcooked Plus and Minecraft. The results demonstrate that SAGE significantly outperforms existing baselines in terms of coverage, efficiency, and bug detection. To promote reproducibility in the field, we are open-sourcing the replication suite used in our experiments.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents SAGE in detail. Section 4 describes our experimental setup and discusses the results. Finally, Section 5 concludes the paper and outlines future research directions.

2 Related Work

Game testing has always played a critical role in the software development lifecycle of games, aiming to ensure stability, completeness, and user experience during both initial release and subsequent updates [9, 10]. With the increasing complexity of modern games and the prevalence of the "game-as-a-service" model, testing approaches have gradually evolved from manual validation to more automated and intelligent solutions. In this section, we review related work from three perspectives: traditional regression testing methods, agent-based automated testing methods, and the applications of LLMs in game agents and testing.

2.1 Traditional Regression Testing Approaches

In the early stages of game testing, the most common practice was to rely on human testers to manually execute and validate tasks[11]. While this approach ensured reliability, it was highly inefficient and did not scale to the vast state spaces of modern games.

To reduce the overhead of repetitive work, [6] proposed *record-and-replay* mechanisms. These methods capture real player interactions and transform them into reusable regression test cases, thereby reducing manual repetition. However, such approaches still relied on testers to identify key interactions, and the recorded traces often contained redundant operations, limiting overall efficiency.

Later, *script-based testing methods* emerged, where testers designed macros or control scripts to simulate player behavior [12, 13]. These methods worked well in fixed scenarios and quests, but script design often required deep domain knowledge of both gameplay mechanics and testing principles. As games became more complex, the cost of maintaining and extending scripts grew rapidly.

Building on this, more advanced forms of automation have also been explored. For instance, GameRTS [8] constructs game state transition graphs through static code analysis and applies RTS techniques to identify affected test cases, thus improving efficiency while reducing maintenance overhead. However, such methods typically assume full access to source code and internal resources—an assumption that is often unrealistic in practical, where testing is usually performed under limited code accessibility [11].

2.2 Agent-Based Automated Testing Approaches

Compared to script-driven testing, which often struggles with scalability and adaptability in dynamic environments, agent-based exploration methods [14–16] employ autonomous agents to simulate diverse player behaviors.

Early-stage works have explored a variety of approaches, including model-based testing for platformers [15] and behavior-tree agents for playability analysis and level evaluation [14, 16], offering enhanced automation and better support for early-stage design and gameplay validation.

Later, reinforcement learning (RL) was introduced into agent-based approaches and has attracted significant attention, as RL agents learn behavioral policies through trial-and-error interactions with the environment, enabling broader coverage and greater adaptability. For example, curiosity-driven intrinsic rewards have been introduced to guide agents toward rarely visited states, improving behavioral diversity and bug detection [7]. [2] developed an RL-based framework that compares agent behavior trajectories across game versions to shift the focus from pure exploration to version-aware correctness. The framework incorporates differences in observed states under identical task conditions into the reward function, enabling the agent to detect potential regressions introduced by updates. Wuji [17] combines deep RL with evolutionary strategies to balance task completion and exploration, further enhancing coverage. Other studies have also attempted to expand exploration capabilities using evolutionary algorithms or preference-driven agents [18].

Nevertheless, these methods primarily focus on discovering new states rather than validating the correctness of updates, and thus lack regression-oriented specificity. Consequently, they often produce large amounts of redundant actions, increasing testing cost and execution overhead. Moreover, RL models require frequent retraining across versions, leading to prohibitive computational expenses in iterative development workflows.

2.3 Large Language Models in Software Testing and Games

LLMs have recently emerged as a transformative force in software testing, enabling a shift from manually designed heuristics to semantics-driven automation [19]. By leveraging deep semantic understanding, reasoning, and multi-step planning capabilities, LLMs can process unstructured artifacts and generate executable test assets, thereby redefining automation across the entire software testing lifecycle. Recent studies, collectively referred to as LLM4TEST [19], have explored the role of LLMs throughout the software testing lifecycle.

Unit Test Case Generation. Unit test case generation automates the creation of test cases for verifying individual functions or methods, serving as the foundation of automated testing. [20] fine-tuned pre-trained transformers on Java corpora to translate focal methods into tests, [21] enhanced generation with assertion-based prompts in A3Test, [22] proposed adaptive prompts for ChatGPT in ChatUniTest, and [23] integrated mutation testing feedback to refine generated cases.

Test Oracle Generation. Test oracle generation focuses on producing expected outputs or assertions that verify correctness in generated tests. [24] fine-tuned T5 for assertion synthesis, [25] demonstrated dual-language pre-training to improve oracle precision, and [26] applied

retrieval-augmented prompting to extract semantically similar examples, achieving higher oracle accuracy without requiring fine-tuning.

System-Level Test Input Generation. System-level input generation targets the testing of interactive systems, where LLMs are applied to generate semantic actions or valid inputs. [27] developed QTypist to generate context-aware GUI inputs, [28] introduced GPTDroid to conduct mobile testing as dialogue-based interaction, and [29, 30] explored fuzzing with LLMs to expose edge cases in deep-learning frameworks.

Bug Analysis and Debugging. Bug analysis and debugging involve using LLMs to interpret, localize, and reproduce software bugs. [31] proposed iTiger for automatic bug title generation, [32] developed Detect–Localize–Repair for bug localization and fixing with CodeT5, [33] explored self-debugging mechanisms where LLMs correct their outputs using runtime feedback, and [34] applied prompting to replay Android crashes automatically.

Program Repair. Program repair applies LLMs to automatically generate or refine code patches for defective software. [35] used GPT-based repair for JavaScript, [36] leveraged retrieval-based bug–fix examples to improve patch accuracy.

In summary, the natural language understanding capabilities of LLMs enable more intelligent automation by integrating richer contextual information into the testing workflow. Inspired by this, and adapting to the prevalent gray-box settings in game testing, our work fully leverages the context tailored in the game testing, and then utilizes the semantics as a medium to connect the entire lifecycle of test case generation, optimization, and selection.

2.4 Position of This Study

In this study, we position our work within a gray-box testing scenario—a realistic middle ground between white-box and black-box settings. In such scenarios, the source code and internal implementation details are inaccessible, yet the system typically exposes a limited degree of structured interaction interfaces, such as APIs, event callbacks, and runtime logs. Testers can observe runtime information (e.g., object states, player actions, and event traces) but cannot directly analyze source-level artifacts such as code differences or class hierarchies. This setting aligns with practical practice: in commercial game development, testing teams are often organizationally separate from development teams. Testers usually have access to certain high-level APIs and detailed runtime logs for testing purposes, while underlying scripts and resource files remain closed due to intellectual property protection and modular development workflows [11, 37].

Under this premise, despite notable progress, several limitations remain in existing studies: Traditional regression testing methods have reduced manual effort but still suffer from the high cost of script maintenance and dependence on source code, which limits their applicability. Agent-based approaches—particularly RL agents—offer strong automation and coverage capabilities but mainly emphasize broad exploration rather than update validation, resulting in redundant actions and high retraining costs. The emergence of LLMs brings new opportunities, as they excel at handling semantic and unstructured information and can further enhance automation; however, their potential in regression testing has not yet been fully explored.

Based on these observations, we propose SAGE, a unified gray-box regression testing framework that bridges the semantic reasoning capabilities of LLMs. The framework eliminates the reliance on manual recording and script-based testing by enabling semantic-driven exploration for efficient and goal-oriented test case generation. It further enhances test case quality through a multi-objective optimization process that balances coverage, cost, and behavioral diversity, and achieves adaptive regression validation via a Update-Aware Test Case Prioritization mechanism that leverages semantic information extracted from natural-language update logs. Notably, SAGE operates solely on log-level information—such as player actions, in-game event traces, and system runtime logs—without requiring any access to source code or internal assets. This design allows the framework to achieve end-to-end automation, high scalability, and semantic adaptability, aligning with the practical constraints of industrial gray-box testing environments.

3 Proposal

3.1 Overview

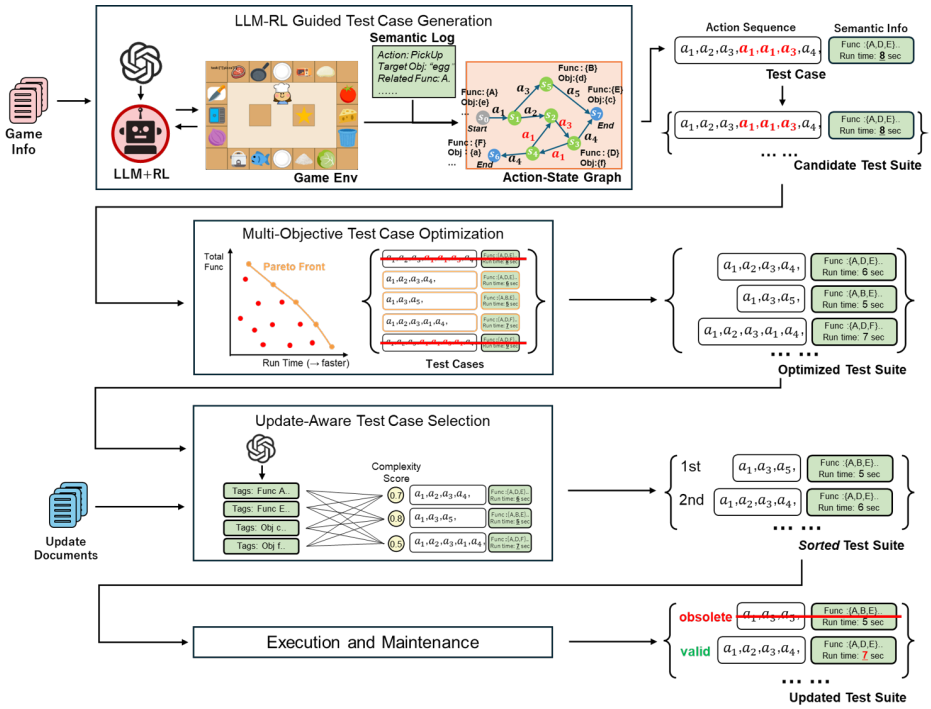

Fig. 1 Overview of SAGE. The framework consists of four main stages: (1) LLM-RL guided test case generation, (2) multi-objective test case optimization, (3) update-aware test case selection, and (4) test case execution and maintenance. These stages together enable automated, scalable, and update-sensitive regression testing in complex game environments. For better visibility and easier understanding, the state information (s) contained in test cases is omitted in this figure.

This section introduces SAGE, a unified gray-box regression testing framework. As illustrated in Figure 1, SAGE operates as a semantic-driven closed-loop system where each stage logically builds upon the output of the last, progressively refining test suite from raw exploration data into a prioritized execution list.

The process begins with LLM-RL Guided Test Case Generation. This stage constructs the foundational test assets by using large language models to interpret task goals and guide reinforcement learning agents in structured exploration. This process results in a comprehensive state–action transition graph, which encapsulates a vast, raw pool of all discoverable candidate test cases.

Given the potential scale and inherent redundancy of this initial pool, the framework immediately proceeds to Multi-Objective Test Case Optimization. This stage acts as a powerful filter, systematically evaluating every path from the graph along competing axes of cost, coverage, and behavioral rarity. By selecting only the Pareto-optimal paths, it distills the raw graph into a compact, high-value, and representative static subset of test cases, which forms the core regression test suite.

To efficiently deploy this repository for a specific version update, the Update-Aware Test Case Selection stage then dynamically prioritizes this static subset. This module uses an LLM to extract semantic tags from natural language update logs. It then ranks the test cases from the static subset based on their semantic relevance to these changes and their intrinsic functional complexity, producing a targeted execution list focused on the most at-risk functionalities.

Finally, the Test Case Execution and Maintenance stage executes this prioritized list on the latest game version. The outcomes (e.g., valid or obsolete) are captured and fed back to update the metadata within the foundational state–action transition graph, ensuring the entire test suite continuously adapts to the evolving game.

3.2 LLM-RL Guided Test Case Generation

As the starting point of the SAGE pipeline, this stage initializes the foundational test assets that capture exploratory behaviors and environment interactions. These assets serve as the raw material for deriving executable test cases. To enable efficient and scalable generation, the stage integrates LLMs, RL, and graph-based modeling: LLMs provide reliable action priors to guide RL exploration, while RL expands and diversifies the behavioral space under semantic constraints. The resulting state–action transition graph abstracts trajectories into a unified, semantically annotated structure, supporting the compositional generation of diverse and meaningful test cases.

3.2.1 Seed Trajectory Generation with LLMs

This module begins by prompting an LLM to generate seed trajectories that are capable of accomplishing a given task. Based on natural language task descriptions and contextual knowledge (e.g., interactable objects, map layout, game rules), the LLM is instructed to simulate the actions a player might reasonably perform to complete the task. Each seed trajectory is a sequence of executable actions (e.g., move, pick up, interact), which is encoded into transition tuples (s_t, a_t, s_{t+1}), where s_t and s_{t+1} denote game states and a_t is the executed

action. These transitions form the initial structure of the state–action transition graph, outlining the behavioral space surrounding the task.

3.2.2 Policy Learning and Guided Exploration

Although LLM-generated trajectories are generally task-effective, they are limited in behavioral diversity and expensive to obtain. To enhance state space coverage, we introduce RL agents to perform environment-driven exploration. Specifically, we use behavior cloning on the LLM-generated seed trajectories to train an initial policy $\pi_0(a \mid s)$, which serves as a strong prior for the RL agent and avoids inefficient exploration from random initialization.

Subsequently, the agent explores the environment using a composite reward function:

$$r(s_t, a_t) = r_{\text{goal}}(s_t) + \frac{1}{n(s_t)},$$

where $r_{\text{goal}}(s_t)$ denotes the terminal reward for reaching a goal state, $n(s_t)$ is the number of times state s_t has been visited. This formulation encourages the agent to prioritize novel states while still completing the task, thereby increasing trajectory diversity.

3.2.3 Graph Construction and Test Case Derivation

All observed transitions during exploration are aggregated into a state–action transition graph $\mathcal{G} = (S, A, E)$, where:

- $S = \{s_0, s_1, \dots\}$: the set of observed game states,
- $A = \{a_0, a_1, \dots\}$: the set of available actions,
- $E \subseteq S \times A \times S$: transitions, where $e = (s, a, s')$ indicates that action a in state s leads to state s'.

Each edge in the graph is annotated with metadata used for subsequent optimization and RTS, such as estimated execution time, the number of interacted objects, and involved scenes (see 3.3.1 for details). These data are obtained from debugging runtime logs generated during interactions, where the log contents are predefined during the game development phase for testing purposes.

A candidate test case is defined as a path τ from the initial state s_0 to a goal state $s_g \in S_{\text{goal}}$:

$$\tau = (s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} s_n), \quad \text{where } s_n \in S_{\text{goal}}.$$

These candidate paths form the input to the multi-objective optimization and update-aware selection modules.

3.3 Multi-Objective Test Case Optimization

In this stage, we compress the large number of candidate test cases generated from the state–action graph based on their associated semantic information, producing a subset that is both representative and execution-efficient. To avoid the information loss caused by collapsing

test cases into a single scoring function, each test case is encoded as a multi-dimensional feature vector based on different semantic metrics. We then apply a multi-objective optimization approach, selecting the set of non-dominated paths on the Pareto front to retain diverse behavioral semantics while eliminating redundant cases.

3.3.1 Objective Formulation and Metric Definition

Inspired by prior work on multi-objective test case optimization [38–40], and adapted to the specific requirements of game testing under limited code accessibility, we define three categories of objectives: cost, coverage, and rarity. Path length and execution time reflect resource cost; coverage metrics capture test value; and we introduce a novel n-gram rarity metric to prioritize paths with potentially high bug triggering capability.

Given a candidate path τ_i, we define:

(1) Cost-related Objectives:

- *Path Length* T_i: number of actions in the path;
- *Execution Time* R_i: estimated time required to execute the path.

(2) Coverage-related Objectives:

- *State Coverage* C_i: number of unique states visited;
- *Action Diversity* A_i: number of distinct actions in the path;
- *Object Diversity* O_i: number of distinct in-game objects interacted with;
- *Scene Coverage* S_i: number of unique map regions visited;
- *UI Component Coverage* U_i: number of distinct UI components triggered.

(3) Rarity Objective:

- *n-gram Rarity* N_i: rarity score of local action subsequences based on their global frequency across all paths.

These features are encoded as a vector for each path and used in multi-objective selection.

3.3.2 Modeling Behavioral Rarity

In real-world game testing, bugs are often triggered by specific local combinations of states and actions rather than by global path patterns. For instance, transitions such as "switching weapons at the exact moment of skill cooldown completion" or "picking up an item when the inventory is almost full" are typical failure-inducing scenarios.

To capture such critical behaviors, we define the n-gram rarity N_i by computing the inverse frequency of short action subsequences (e.g., 3-grams) across the candidate path pool. This metric favors paths with rare yet plausible behaviors, thereby increasing the likelihood of exposing edge-case bugs without inflating the optimization problem's dimensionality.

3.3.3 Pareto-Optimal Path Selection

Since the objectives are often in conflict (e.g., longer paths cover more but cost more), we adopt a Pareto front-based approach to select test cases. A path τ_i is Pareto-optimal if there is

no other path τ_j such that:

$$\forall k, \quad f_k(\tau_j) \leq f_k(\tau_i) \quad \text{and} \quad \exists k, f_k(\tau_j) < f_k(\tau_i),$$

where f_k denotes the k-th objective (e.g., cost, coverage, or rarity).

The final optimized test suite $\mathcal{P}^*$ is composed of all Pareto-optimal paths, representing an effective balance among competing objectives.

3.4 Update-Aware Test Case Selection

This stage prioritizes test cases based on their semantic relevance to updates and their functional complexity. We use an LLM to extract structured semantic tags from update logs and compare them with the semantic information contained in each test case to compute relevance. A complexity score is calculated based on interaction diversity, and both signals are combined to produce update-aware execution priorities.

3.4.1 Semantic Change Extraction from Update Logs

Given an update document U (e.g., update logs), we use an LLM to extract a set of structured semantic tags:

$$\mathcal{K} = \{k_1, k_2, \ldots, k_n\},$$

where each k_i denotes a human-readable descriptor of a changed gameplay feature (e.g., "crafting system", "item logic", "time cost adjustment"). These tags act as lightweight proxies for functional changes and help localize affected behaviors without requiring code access.

3.4.2 Test Case Prioritization

Each test case $t_j \in \mathcal{T}$ is annotated with semantic metadata (actions, objects, scenes, UI components, and states) and encoded as a vector $\mathbf{v}_{t_j}$, using a pre-trained Sentence-BERT model. The semantic similarity between a test case and the update is measured via cosine similarity:

$$\text{sim}(t_j, \mathcal{K}) = \max_{k_i \in \mathcal{K}} \cos\left(\mathbf{v}_{t_j}, \mathbf{v}_{k_i}\right).$$

While semantic similarity captures direct relevance, many bugs are caused by the interaction of new features with existing complex logic. To reflect the intrinsic test value of each case, we define a Semantic Complexity Score based on the diversity of interactions. Specifically, we normalize the counts of distinct actions, objects, scenes, UI components, and states into the range $[0, 1]$, and compute their average. To prevent long paths from inflating the score, we further divide by the normalized path length:

$$\text{SCS}(t_j) = \frac{\hat{A}_j + \hat{O}_j + \hat{S}_j + \hat{U}_j + \hat{G}_j}{\hat{L}_j},$$

where $\hat{A}_j, \hat{O}_j, \hat{S}_j, \hat{U}_j, \hat{G}_j$ denote the normalized counts of actions, objects, scenes, UI components, and states, respectively, and $\hat{L}_j$ denotes the normalized path length.

The final prioritization score combines both semantic relevance and complexity:

$$\text{Score}(t_j) = \lambda \cdot \text{sim}(t_j, \mathcal{K}) + (1 - \lambda) \cdot \frac{\text{SCS}(t_j)}{\max_{t \in \mathcal{T}} \text{SCS}(t)},$$

where $\lambda \in [0, 1]$ balances update relevance and functional complexity. This hybrid strategy ensures that high-priority cases not only cover newly changed content but also retain a potential to uncover critical bugs.

3.5 Test Case Execution and Maintenance

To ensure efficient regression testing, we adopt a lightweight incremental maintenance strategy. Each test case is re-executed on the latest game version and classified based on outcome: If the original goal is still achievable, the test case is marked as *valid* and its associated metadata in the test repository is updated accordingly. Otherwise, it is marked as *obsolete* and flagged for manual review or regeneration.

4 Evaluation

We aim to evaluate our proposed framework through three research questions:

- **RQ1 (Effectiveness):** How well does SAGE detect and diversify bugs compared to baseline methods?
- **RQ2 (Efficiency):** How efficient is SAGE in performing regression testing during version updates?
- **RQ3 (Ablation Analysis):** What are the respective contribution of the multi-objective optimization and update-aware prioritization modules?

4.1 Experimental Setup

4.1.1 Game Environments

(a) Overcooked Plus (b) Minecraft

Fig. 2 Illustration of game environments used in evaluation.

We conduct experiments in two game environments: Overcooked Plus and Minecraft. *Overcooked Plus.* Overcooked Plus [41] is an open-source reimplementation of the commercial game *Overcooked* [42]. It preserves the original's high task complexity while enhancing operational constraints and system controllability. This environment is particularly suitable for evaluating the execution accuracy and planning capability of testing methods in rule-driven games. In this environment, players must complete sequential operations such as chopping, cooking, plating, and serving. The rules are strict and deterministic; for example, each workstation can only hold one item, and mistakes require explicit correction. The environment is fully observable, with the state and position of all objects represented in structured form, which facilitates behavioral modeling and analysis. We designed 37 tasks and 2 version updates (V1→V2 and V2→V3). Specifically, V1 contains 10 tasks, and V2 extends to 27 tasks. Both updates not only introduce new game tasks but also include bug fixes and modifications to interaction logic. During development, we conducted systematic and unbiased bug collection, ensuring that the recorded issues—spanning logic errors, interaction conflicts, UI or rendering issues, and occasional physics anomalies—reflect a representative distribution of real-world problems encountered during gameplay. Reproducible triggers were constructed for each bug to support rigorous regression testing. This environment emphasizes precise sequential planning and error tolerance, making it well-suited for assessing the breadth of behavioral coverage and robustness of testing methods in rule-driven settings.

Minecraft. Minecraft [43] is a highly open-ended sandbox game with extremely rich interaction mechanisms and a vast state space, and has been widely adopted in AI research. This environment is suitable for evaluating the generalization ability and adaptive strategies of testing methods in open-world and exploratory tasks. The Minecraft environment used in our study is based on a custom mod currently under development. The tasks within this mod are inspired by the original achievement system, covering multiple interaction types such as resource collection, crafting, and combat. To ensure compatibility across baselines, complex operations are abstracted into parameterized commands (e.g., "mine," "craft," "move"), which are then mapped to atomic low-level actions. The environment is also fully observable, exposing key variables such as character position, inventory state, available resources, and tool usage. In this environment, we define 50 tasks and two version updates (V1→V2 and V2→V3), covering action space extensions, item accessibility adjustments, and quest logic modifications. All bugs recorded during development are retained and made reproducible through trigger mechanisms to ensure the authenticity and comparability of results. The embedded bugs mainly involve quest logic and trigger errors, resource or inventory inconsistencies, collision or pathfinding failures, and performance anomalies. This environment emphasizes long-term planning and strategic flexibility, making it suitable for evaluating the exploration breadth, behavioral diversity, and version adaptability of testing methods in open-world settings.

4.1.2 Evaluation Metrics

For effectiveness (RQ1), we consider *Bug Count*, *Unique Bugs*, and *Unique States*. *Bug Count* measures the total number of bug activations under a fixed testing budget. A bug occurrence is defined as the triggering of a pre-defined bug (e.g., illegal state transition, resource overflow, inconsistent game logic). Bug Count is reported as a global cumulative

value across all episodes. *Unique Bugs* records the number of distinct bug IDs triggered at least once. Each bug is assigned a unique ID, and each is counted only once, even if repeatedly triggered across multiple episodes. *Unique States* represents the number of distinct environment states visited during testing, reflecting the behavioral diversity of generated test cases. In practice, environment states are encoded as structured vectors (in Overcooked Plus: all object positions and player states; in Minecraft: player position, inventory contents, and local block configuration). Each state vector is hashed for efficient deduplication. Unlike Bug Count and Unique Bugs, Unique States is first counted on a per-episode (i.e., per quest) basis and then aggregated across all episodes to obtain the final cumulative result. In addition, we report *Reward* and *Success Rate* to reflect the behavioral depth of test execution. *Reward* represents the cumulative in-game reward obtained per episode during each test case, while *Success Rate* denotes the proportion of test cases that successfully complete their designated tasks.

For efficiency (RQ2), we record both *Duration*, the total wall-clock time required to execute all test cases of a method[1], and *Steps*, the cumulative number of action steps performed during testing.

4.1.3 Comparison Methods

We compare against five representative methods, described below:

(i) Random [44, 45]. A general-purpose exploration baseline based on uniform random action sampling over all available actions, with each candidate action assigned equal probability. Invalid actions are not excluded. This serves as a lower bound for uninformed exploration.

(ii) PPO [46]. As a state-of-the-art general-purpose exploration agent, we adopt the Stable-Baselines3 [47] implementation. Models are pre-trained for 100,000 steps with $\gamma = 0.95$, learning rate 3×10^{-4}, and batch size 256, and then stored in `data/models`. During inference, we load the corresponding model and execute it in stochastic mode (`deterministic = False`) for the prescribed number of steps, consistent with other baselines.

(iii) diff-Qlearning [2]. In contrast to the general-purpose exploration methods above, diff-Qlearning is a regression-specific RL approach proposed by [2], specifically designed for detecting potential regressions in game environment. The method trains a Q-learning agent across different software versions, leveraging *differential behavior signals* as implicit supervision to guide exploration toward areas likely to contain regressions. Unlike policy-based methods such as PPO, diff-Qlearning treats training trajectories themselves as test cases. Each episode is executed under both the old and new versions of the game, and behavioral discrepancies—such as differences in rewards, state transitions, or violation of pre-defined rules—are used to reinforce states that may indicate bugs introduced during version updates. Following the configuration described in the original paper, we implement diff-Qlearning using a learning rate $\alpha = 0.2$, discount factor $\gamma = 0.99$, and an initial exploration rate $\epsilon = 0.5$, which decays every 100 steps until reaching a minimum of 0.01. The total training step budget is kept consistent with other baselines to ensure fair comparison.

[1] All experiments were conducted on a workstation equipped with an AMD Ryzen 9 9950X3D CPU, 64 GB RAM, and an NVIDIA RTX 5090 GPU running Ubuntu 22.04 under Windows 11 WSL.

(iv) Human-recorded test cases [6]. To align with practical settings, we also include human-recorded test cases as high-quality human baselines [11]. Three experienced testers participated, each with 2–3 years of professional experience in game QA or development and over 200 hours of gameplay experience in the original versions of *Overcooked* and *Minecraft*. Each tester recorded five trajectories per task using keyboard and mouse, resulting in approximately 1,800 trajectories in total. On average, each version update required about 6 hours of recording per participant.

(v) SAGE (ours). For each environment, we prompt LLM to generate 20 seed trajectories per task. These seeds are used to train a behavior cloning policy, which initializes a PPO agent for guided exploration (using the same PPO configuration as above but with the task-specific reward design introduced below). Candidate paths are evaluated using multi-objective optimization over cost, coverage, and rarity, with rarity measured via 2-gram subsequences. During experiments, we record different degrees of RTS proportion (from 10% to 90%), and unless otherwise specified, we use SAGE (top 50%) as the default configuration for discussion to ensure consistency and clarity. *Ablation.* We further evaluate four configurations of SAGE to analyze the contribution of its key components: the full version (multi-objective optimization + update-aware test case prioritization), w/o multi-opt (remove the optimization module), w/o RTS (remove the update-aware prioritization module), and w/o both (disable both modules while retaining all generated outputs). All ablation settings follow identical experimental configurations as other methods. We consistently used GPT-4o as the LLM model throughout the entire experimental process.

To ensure fair and consistent evaluation, all learning-based methods (PPO, diff-Qlearning, and SAGE) share the same action and observation space, random seed (`seed=42`). For all reinforcement-learning-based methods, each episode and task follow a fixed step budget to control exploration cost. In Overcooked Plus, each episode is capped at 100 steps and each task at 5000 steps; in Minecraft, 75 steps per episode and 3000 per task. In Overcooked Plus, the reward design assigns +1000 for each stage goal, +10000 for completing the overall quest, and -0.1 per action step. In Minecraft, the design assigns -0.5 per action step, +10 for intermediate goals (e.g., obtaining relevant items), and +100 for completing the quest. Each method is executed independently 10 times (except Human-recorded tests), and results are reported as the mean with 95% confidence intervals. A two-tailed t-test is applied to assess statistical significance.

4.2 Experimental Results

4.3 RQ1: Effectiveness

The experimental results are summarized in Table 1 and Table 2. In the Overcooked Plus environment, SAGE detected an average of 37.5 and 57.7 unique bugs in the two regression phases, closely approaching human-recorded trajectories (40.3 and 65.7) and far exceeding all automated baselines. For instance, PPO and diff-Qlearning identified only around 19.3 and 28.4 unique bugs in the V1→V2 phase, and 26.4 and 38.3 in the V2→V3 phase—roughly half of SAGE's diversity. Notably, even though the number of unique states explored by SAGE

Table 1 Comparison of methods on V1→V2 and V2→V3 in Overcooked Plus (mean with standard deviation). SAGE conducts experiments on different RTS proportion and compiles the results.

Version	Method	RQ1						RQ2	
		Episodes	Bug Count	Unique Bugs	Unique States	Reward	Success Rate	Duration (s)	Total Steps
V1→V2	RANDOM	500.4 (±0.5)	5313.8 (±33.8)	23.9 (±2.0)	6920.1 (±161.1)	-491.2 (±12.1)	0.00 (±0.00)	14.5 (±0.2)	50000.0 (±0.0)
	PPO	565.0 (±37.1)	25835.7 (±4078.3)	19.3 (±3.1)	218.6 (±50.1)	171.8 (±86.9)	0.18 (±0.08)	69.3 (±1.3)	50000.0 (±0.0)
	diff-Qlearning	523.2 (±21.4)	4722.2 (±694.1)	28.4 (±3.1)	2607.4 (±208.3)	-42.3 (±13.2)	0.00 (±0.00)	35.1 (±0.5)	50000.0 (±0.0)
	HUMAN	80.0	743.7	40.3	1490.0	1162.9	0.99	14.9	3038.7
	SAGE (top 10%)	119.3 (±8.0)	974.4 (±84.9)	33.8 (±2.1)	1832.5 (±128.7)	1138.8 (±22.4)	1.00 (±0.00)	1.3 (±0.1)	4419.1 (±289.1)
	SAGE (top 30%)	357.0 (±24.4)	3000.9 (±244.0)	36.9 (±2.4)	2992.5 (±178.6)	1153.3 (±14.8)	1.00 (±0.00)	3.8 (±0.3)	13004.9 (±893.0)
	SAGE (top 50%)	594.3 (±40.5)	5038.5 (±415.9)	37.5 (±2.3)	3526.6 (±200.2)	1160.3 (±11.6)	1.00 (±0.00)	6.2 (±0.5)	21189.3 (±1474.4)
	SAGE (top 70%)	832.1 (±56.7)	7102.7 (±581.5)	38.2 (±2.2)	3769.6 (±213.5)	1163.0 (±10.9)	1.00 (±0.00)	8.5 (±0.6)	28913.3 (±2073.8)
	SAGE (top 90%)	1069.8 (±73.1)	9283.6 (±749.6)	38.3 (±2.2)	3840.3 (±213.2)	1167.8 (±10.0)	1.00 (±0.00)	10.8 (±0.9)	36276.6 (±2700.7)
V2→V3	RANDOM	1850.6 (±0.7)	10554.5 (±64.5)	35.4 (±1.3)	21531.3 (±186.6)	-415.0 (±5.8)	0.00 (±0.00)	58.1 (±0.4)	185000.0 (±0.0)
	PPO	1975.0 (±57.6)	62712.9 (±10990.9)	26.4 (±2.7)	537.6 (±67.6)	76.6 (±43.6)	0.09 (±0.04)	253.2 (±3.7)	185000.0 (±0.0)
	diff-Qlearning	1955.1 (±73.3)	11263.8 (±1264.0)	38.3 (±2.4)	7484.2 (±478.2)	-27.0 (±8.8)	0.00 (±0.00)	135.9 (±0.6)	185000.0 (±0.0)
	HUMAN	224.7	2204.3	65.7	4972.0	1144.0	0.99	22.8	9134.3
	SAGE (top 10%)	388.5 (±16.0)	3072.4 (±117.4)	53.7 (±1.6)	5449.1 (±160.5)	1167.5 (±13.3)	1.00 (±0.00)	3.3 (±0.1)	14404.7 (±584.4)
	SAGE (top 30%)	1164.9 (±47.8)	9220.0 (±337.2)	56.8 (±1.7)	8494.0 (±245.4)	1168.3 (±12.3)	1.00 (±0.00)	9.8 (±0.5)	42432.3 (±1755.3)
	SAGE (top 50%)	1941.0 (±79.5)	15173.9 (±617.9)	57.7 (±2.0)	9874.1 (±254.9)	1164.1 (±12.2)	1.00 (±0.00)	16.1 (±0.9)	69284.9 (±2837.6)
	SAGE (top 70%)	2717.5 (±111.2)	20975.7 (±911.8)	58.6 (±2.0)	10539.0 (±269.0)	1158.9 (±13.4)	1.00 (±0.00)	22.1 (±1.2)	95129.2 (±4073.9)
	SAGE (top 90%)	3493.9 (±143.1)	26602.6 (±1055.3)	58.6 (±2.0)	10793.0 (±279.5)	1153.8 (±15.2)	1.00 (±0.00)	27.9 (±1.5)	119422.9 (±5084.1)

is lower than Random exploration, it consistently achieves higher unique-bug diversity and a perfect success rate (1.0).

A similar pattern is observed in the Minecraft environment, which presents a significantly larger and more complex state space. While PPO and diff-Qlearning triggered over 20K total bug activations, they revealed fewer than 26 unique bugs, indicating repetitive activation of shallow, low-impact issues. In contrast, SAGE uncovered 39 and 42 unique bugs in the two regression phases—about 1.6× more than the automated baseline—while maintaining competitive state diversity (20K–70K). These results show that the proposed framework

Table 2 Comparison of methods on V1→V2 and V2→V3 in Minecraft (mean with standard deviation). Rewards are normalized by Episodes (Reward/Episode).

Version	Method	RQ1						RQ2	
		Episodes	Bug Count	Unique Bugs	Unique States	Reward	Success Rate	Duration (s)	Total Steps
V1→V2	RANDOM	1652.9 (±8.1)	21726.3 (±203.6)	22.0 (±2.7)	5249.3 (±135.7)	-16.8 (±0.8)	0.06 (±0.01)	61.7 (±3.1)	120000 (±0)
	PPO	1654.3 (±5.6)	21721.9 (±267.1)	23.3 (±2.4)	5623.3 (±82.2)	-15.4 (±0.5)	0.06 (±0.05)	269.3 (±12.8)	120000 (±0)
	diff-Qlearning	1658.4 (±8.3)	21957.0 (±209.7)	23.3 (±2.2)	5524.9 (±54.5)	-15.1 (±0.8)	0.07 (±0.01)	389.0 (±19.4)	120000 (±0)
	HUMAN	200	783	41.3	1789	142.1	1.00	42.7	3857
	SAGE (top 10%)	200.8 (±26.8)	9838.4 (±1887.6)	38.2 (±2.7)	7313.0 (±1549.0)	104.8 (±14.9)	1.00 (±0.00)	53.8 (±2.7)	19799.2 (±3073.4)
	SAGE (top 30%)	664.6 (±57.6)	20414.2 (±3474.6)	38.8 (±1.6)	16402.8 (±2781.7)	116.2 (±10.4)	1.00 (±0.00)	94.2 (±4.7)	49426.0 (±6441.9)
	SAGE (top 50%)	1069.4 (±172.7)	25031.6 (±6397.6)	39.0 (±2.0)	20201.6 (±5250.4)	122.7 (±21.7)	1.00 (±0.00)	134.6 (±6.7)	66094.0 (±12996.1)
	SAGE (top 70%)	1519.0 (±186.1)	29428.0 (±7190.8)	39.1 (±2.0)	22185.4 (±3425.0)	126.9 (±15.4)	1.00 (±0.00)	161.5 (±8.0)	80558.2 (±13535.1)
	SAGE (top 90%)	1974.0 (±259.7)	32151.6 (±6658.4)	39.2 (±2.0)	23280.6 (±4514.0)	130.5 (±19.8)	1.00 (±0.00)	182.7 (±9.1)	91234.4 (±14579.7)
V2→V3	RANDOM	3338.6 (±11.0)	43455.3 (±208.1)	24.0 (±1.8)	10510.4 (±144.6)	-15.0 (±0.6)	0.07 (±0.05)	123.5 (±6.3)	240000 (±0)
	PPO	3361.1 (±12.7)	43713.1 (±578.7)	25.7 (±1.2)	11229.3 (±113.0)	-12.9 (±0.4)	0.08 (±0.06)	538.7 (±26.3)	240000 (±0)
	diff-Qlearning	3362.6 (±15.1)	44001.9 (±335.5)	25.3 (±2.3)	10941.6 (±82.1)	-13.0 (±0.7)	0.08 (±0.05)	777.9 (±38.9)	240000 (±0)
	HUMAN	400	1896	45.4	4523	149.3	1.00	85.3	9836
	SAGE (top 10%)	489.8 (±58.6)	31827.2 (±3974.6)	41.8 (±1.4)	24387.6 (±2700.7)	111.7 (±13.5)	1.00 (±0.00)	53.8 (±2.7)	58063.4 (±7126.4)
	SAGE (top 30%)	1604.0 (±95.0)	65699.0 (±6657.8)	42.0 (±1.2)	56541.4 (±3207.5)	125.5 (±7.3)	1.00 (±0.00)	94.2 (±4.7)	143468.8 (±9028.6)
	SAGE (top 50%)	2562.8 (±243.9)	78379.0 (±5634.8)	42.0 (±1.2)	69782.4 (±6312.6)	133.3 (±13.8)	1.00 (±0.00)	134.6 (±6.7)	188592.0 (±14826.1)
	SAGE (top 70%)	3647.2 (±289.0)	87927.6 (±7485.1)	42.2 (±1.4)	74639.4 (±1929.9)	139.5 (±10.6)	1.00 (±0.00)	161.5 (±8.0)	223544.6 (±20127.8)
	SAGE (top 90%)	4735.8 (±463.4)	95531.6 (±12404.4)	42.2 (±1.4)	76924.8 (±7286.2)	143.7 (±14.9)	1.00 (±0.00)	182.7 (±9.1)	250517.0 (±21743.4)

achieves robust generalization and semantic adaptability even under open-world conditions. Through the collaboration of LLM and RL, SAGE can stably complete tasks while leveraging its state–action graph to infer additional feasible paths, effectively generating more diverse and semantically rich test cases from limited exploration. This structured exploration paradigm grants it a unique advantage in uncovering deep logic regressions.

In some cases, we observed that SAGE failed to capture certain shallow bugs that were detected by diff-Qlearning or Random. Two main factors contribute to this behavior: (1) During graph construction, although exploration rewards encourage broad coverage, the dominant

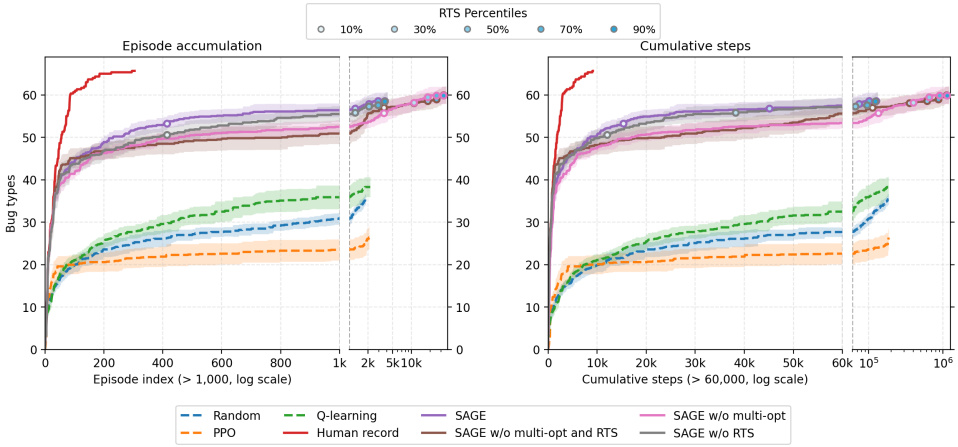

Fig. 3 Ablation study of SAGE in Overcooked Plus. The figure shows the results of V2→V3.

task-oriented rewards still bias the agent toward goal completion. As a result, some shallow or off-task states are underexplored, leading to missed edge-case bugs (e.g., transient state-switching anomalies). (2) During test case extraction from the graph, path search depth and breadth must be constrained for computational feasibility. In complex scenarios such as Minecraft V2→V3, the task length and world openness lead to very large graph structures, forcing pruning that excludes certain peripheral states. Consequently, a portion of off-task test trajectories—while visited during exploration—are omitted from the final test set. This trade-off becomes more pronounced as task complexity increases. Overall, these limitations reflect a balance between task-oriented depth and state-space breadth: SAGE prioritizes deep exploration and high-value bug detection at the cost of reduced coverage of non-critical states.

Taken together, these results reveal a key insight: the "quantity" and "depth" of bug detection are independent yet equally important dimensions. While brute-force exploration may yield higher bug counts, truly effective regression testing depends on semantic diversity and coverage depth. Leveraging LLM-guided semantics and multi-objective optimization, SAGE strikes a balance between efficiency and quality—systematically exploring diverse behaviors while focusing on representative, high-value trajectories. Furthermore, its consistently high task completion rate enables the system to traverse deep interaction chains and uncover latent logic regressions across modules and states. From a practical standpoint, this quality-oriented testing paradigm better aligns with the fundamental goal of regression testing. The essence of regression testing lies in verifying whether a system can preserve the correctness and consistency of its core functionalities after updates or evolution. Therefore, rather than pursuing broad yet shallow bug activations, it is more crucial to conduct semantically deep explorations of critical functional chains to uncover complex logical or interactive regressions. By prioritizing depth over breadth, this strategy yields testing outcomes that more faithfully reflect the real quality risks of an evolving system. It avoids the redundant outputs of massive, repetitive,

and low-impact issues commonly produced by traditional automation, thus achieving a more efficient and targeted regression verification process.

Summary of RQ1:

SAGE exhibits near human-level bug diversity and robust generalization across both environments. It detects approximately 1.6× more unique bugs than the automated baselines and performs only slightly below human experts. Unlike baselines that mainly uncover shallow or repetitive issues, the consistently high task success rate enables SAGE to reach late-stage interactions and expose complex cross-module regressions. Overall, this "goal-oriented testing approach that balances depth and breadth" aligns more closely with the essence of regression testing—prioritizing the verification of critical functional chains after system evolution—thereby providing a more accurate and efficient assessment of real quality risks than approaches that merely pursue wide but shallow coverage.

4.4 RQ2: Efficiency

As shown in Table 1 and Table 2, SAGE consistently demonstrates significant efficiency advantages over all automated baselines across both version updates (V1→V2 and V2→V3), second only to Human Record. In the Overcooked Plus environment, SAGE completes each regression cycle using roughly 21–69K steps, whereas PPO, diff-Qlearning, and Random exhaust their full 50K–185K step budgets. In terms of duration, SAGE requires only about 6–16 seconds per cycle, compared with 14–69 seconds for PPO and diff-Qlearning. Despite the much smaller interaction budget, SAGE still achieves higher unique-bug diversity and maintains a 100% task success rate, indicating that its exploration trajectories are both efficient and representative.

The trend is similar in the Minecraft environment, which features a larger state space, higher task complexity, and greater exploration cost. SAGE completes the regression process within approximately 66–189K steps, while all RL-based baselines require the full 120–240K steps. During the V1→V2 transition, SAGE takes about 134 seconds on average, compared to 269 seconds for PPO and 389 seconds for diff-Qlearning. During V2→V3, it completes the process in roughly 134 seconds, while PPO and diff-Qlearning require 538 and 778 seconds, respectively. Overall, SAGE reduces execution time by roughly 75–90% compared with automated baselines while maintaining superior bug diversity and success rates, demonstrating strong scalability and efficiency in large-scale environments.

The differences in duration among methods mainly arise from their respective planning paradigms. Traditional RL methods such as PPO and diff-Qlearning adopt an online planning approach, where policies must be queried or updated continuously during testing. For instance, diff-Qlearning performs a Q-table lookup for every action decision and computes differential rewards across two environments, significantly increasing duration. Although PPO exhibits more stable convergence, its reliance on neural policy inference leads to higher decision latency per inference. In addition, the aforementioned duration does not include the training overhead. Especially in regression testing scenarios involving frequent version iterations, PPO

and similar model-dependent inference methods often require retraining or fine-tuning as the system updates, introducing additional time and computational costs that further undermine their practicality.

Although SAGE achieves outstanding efficiency during the execution phase, it incurs a certain amount of preprocessing overhead during graph construction and multi-objective optimization. However, since the constructed state–action graph and optimized test cases can be reused across multiple versions, this offline cost can be effectively amortized. Furthermore, while Human Record achieves the highest raw execution efficiency, the manual recording of gameplay trajectories typically requires several hours per session, limiting scalability in iterative testing pipelines.

To study the impact of the RTS proportion, we examined RTS proportions of 10%, 30%, 50%, 70%, and 90% to explore the trade-off between testing depth and efficiency. Across both Overcooked and Minecraft, small proportions (10%–30%) yield most of the bug diversity while requiring only a fraction of the steps and time. For example, in Overcooked V1→V2, selecting the top 10% of paths produces about 33.8 unique bugs in roughly 4K steps and 1.3 seconds, whereas increasing to 30% achieves around 36.9 unique bugs at 13K steps and 3.8 seconds. Expanding the subset to 70% or 90% increases the step count to 29–36K and duration to 8–11 seconds, yet the number of unique bugs saturates near 38. A similar pattern is observed in the V2→V3 transition and in Minecraft: smaller selections reduce steps and runtime dramatically with modest reductions in unique bugs, while larger selections yield only incremental coverage gains at much higher cost. These results underline that moderate selections (10%–50%) can provide rapid validation or a balanced trade-off between efficiency and coverage, whereas very large proportions offer minimal additional benefit and significant overhead.

Summary of RQ2:

SAGE achieves comparable or higher bug coverage while using only about 10–40% of the interaction steps required by baseline methods and reduces average duration by roughly 75–90%, ranking second only to Human Record. Its efficiency advantage primarily derives from the synergy between graph-based multi-objective optimization and update-aware test case prioritization, which together minimize redundant executions. Although the preparation stage incurs additional overhead, this cost can be amortized across multiple version updates, making SAGE highly efficient for iterative regression testing. Moreover, varying the RTS proportion enables flexible trade-offs between efficiency and coverage, with smaller subsets (10–50%) already achieving near-optimal balance under most conditions.

4.5 RQ3: Ablation Analysis

To evaluate the contribution of the core modules within SAGE, we conducted ablation studies in the Overcooked Plus environment, the results are shown in Fig. 3.

From the perspective of unique bugs coverage, both the optimization and prioritization modules play crucial roles in improving testing diversity and efficiency. Compared to baselines,

SAGE and its variants rapidly cover a large number of unique bugs during the early stages and maintain higher overall coverage across the entire testing process. When the optimization module is removed (w/o multi-opt), the coverage curve saturates quickly, suggesting that redundant paths lead to repeated triggering of the same unique bugs. When the update-aware test case selection module is removed (w/o RTS), the system still achieves good final coverage but requires more episodes to reach comparable levels, indicating lower efficiency in early exploration. In contrast, the complete SAGE consistently outperforms all variants, maintaining both high diversity and fast convergence under limited testing budgets.

The ablation results highlight the distinct yet complementary roles of the two modules. The optimization module ensures that limited testing resources are spent on diverse and representative trajectories rather than redundant ones, directly improving coverage efficiency. Meanwhile, the update-aware test case selection module accelerates early-stage testing by executing the most valuable cases first. In practical regression testing scenarios, this distinction is highly relevant: large-scale periodic testing benefits from optimization to achieve comprehensive long-term coverage, whereas rapid hotfix validation gains from the early efficiency boost provided by prioritization.

Interestingly, the variant without optimization (w/o multi-opt) occasionally achieves a slightly higher final unique bug coverage than the full framework, but only after several times more episodes and interaction steps. This finding suggests that while multi-objective optimization substantially enhances efficiency by reducing redundancy, it may also prune certain rare, low-frequency behaviors. From a practical perspective, this is a reasonable trade-off—efficiency gains are often more valuable than marginal increases in total coverage, especially under tight time budgets. Nevertheless, this observation points to an important future direction: developing adaptive optimization strategies that retain pruning efficiency while selectively preserving long-tail behaviors, enabling both rapid bug discovery and maximal long-term coverage.

Summary of RQ3:

Both the multi-objective optimization and update-aware prioritization modules contribute critically to SAGE's superior performance. Optimization eliminates redundant trajectories and enhances the diversity and efficiency of generated test cases, while prioritization accelerates early bug discovery by executing high-value test cases first. Together, these two modules balance efficiency and coverage, enabling effective and scalable regression testing under limited resources.

4.6 Discussion and Limitations

Beyond SAGE, the value of human-recorded test cases is also noteworthy. Although their coverage range is limited, they consistently exhibited high success rates and well-structured paths, reflecting the intuition and efficiency of experienced testers. This explains why human testing remains indispensable in practical practice and also motivates us to examine where automation still falls short.

Although our method has achieved promising results, several limitations remain that warrant further discussion. First, the effectiveness of RL in complex and high-dimensional game environments remains a fundamental challenge. In some practical scenarios, RL agents may fail to complete tasks or to explore sufficiently diverse state spaces, thereby reducing the quality of the constructed state–action graphs. While prior studies have shown that LLMs demonstrate strong reasoning abilities in complex task decomposition, fully replacing RL with LLM-generated action sequences or relying on LLM self-correction mechanisms remains an open question. As a supplementary solution, incorporating human-recorded trajectories to assist graph construction is feasible in practice. Given the modular design of our framework, such alternatives can be flexibly integrated, but their effectiveness in practical environments still requires further validation.

Second, the multi-objective optimization stage depends on the availability of sufficient configuration and logging information from the game environment. In research prototypes, such information is often directly accessible. However, in more restricted grey-box environments, where internal system details are only partially observable, detailed debugging metrics are rarely available. A potential alternative is to capture screen recordings (e.g., via OBS) or use lightweight scripts to observe gameplay and infer environment changes through frame differencing, thereby augmenting the statistical features. Nevertheless, this approach introduces additional engineering overhead, and its accuracy and scalability remain open challenges.

Third, our framework may incur non-trivial computational costs. In particular, the path extraction step from the constructed graph can become infeasible when the graph grows excessively large in complex games, leading to state explosion and resource exhaustion. This issue may hinder deployment in real-world contexts. The current mitigations include imposing limits on path search depth and maximum path counts, while future work could incorporate further pruning strategies to preserve behavioral diversity while reducing computational overhead. Moreover, although LLMs provide strong reasoning ability, their inference itself also introduces noticeable computational overhead and latency, which should be considered when deploying the framework at scale.

Finally, over long-term iterations, the gradual obsolescence of test cases may result in test suite drift, thereby reducing the effectiveness of regression testing. Therefore, test suite regeneration and manual intervention are still required. Future work should further explore more efficient incremental maintenance mechanisms to minimize redundant computation and ensure the long-term scalability of the framework in high-frequency, live-service game environments.

4.7 Threats to Validity

4.7.1 Internal validity

First, the training of RL agents in SAGE is inherently stochastic, as it depends on random initialization and exploration trajectories. Different random seeds may lead to noticeable performance variations. To mitigate this threat, we execute each experiment with multiple seeds and report averaged results together with standard deviations or confidence intervals.

Second, the performance of baseline methods can be highly sensitive to the design of reward functions and hyperparameter configurations. Suboptimal tuning may result in unfair comparisons and underestimate their true potential. To minimize this risk, we designed competitive reward functions and parameter settings in line with best practices.

Third, our evaluation relies on bug triggers that are either recorded during environment development. Although this design ensures controllable and repeatable bug detection, it may not fully cover the diversity of bugs encountered in real-world practical scenarios.

4.7.2 External validity

A first concern lies in the representativeness of the evaluated environments. Although our experiments cover Overcooked Plus and Minecraft, which represent rule-driven and open-world game genres respectively, and the Minecraft environment, as a custom mod under active development, may have different internal development practices and a resulting bug profile compared to a mature commercial title, they still do not encompass other mainstream types such as first-person shooters, card games, narrative-driven adventures, or large-scale multiplayer titles. As a result, it is difficult to guarantee that our approach will generalize equally well across all categories of real-world games.

The update log used in our experiments was written with reference to publicly available update logs on commercial gaming platforms such as Steam[48]. They include a wide range of update types (new content, gameplay adjustments, bug fixes, etc.), thereby covering the diversity typically seen in real-world update logs. At the same time, they deliberately preserve common characteristics of real logs, such as brevity, ambiguity, and partial information. However, we acknowledge that these manually curated logs may still not capture all real-world variations, as some commercial projects exhibit more fragmented or even misleading update records. In future work, we plan to evaluate the robustness of our framework within actual commercial game development environments.

Finally, our evaluation primarily measures bug detection by counts and diversity without distinguishing the severity or business impact of individual bugs. In practical contexts, however, the ability to prioritize critical bugs may be more important than maximizing the sheer number of detected bugs. This mismatch between academic metrics and practical priorities also poses a potential external validity threat.

5 Conclusion and Future Work

This paper presents SAGE, a unified framework for automated gray-box regression testing in games. Driven by semantic information, the framework leverages LLMs to systematically address three core issues. It solves the Foundation Issue using LLM-guided RL exploration to build a diverse test suite. It addresses the Maintenance Issue via a semantic-based multi-objective optimization to refine the suite into a compact, high-value subset. Finally, it tackles the Selection Issue by using an LLM to interpret update logs and translate changes into dynamic test priorities.

We conducted a comprehensive evaluation in two representative game environments, comparing SAGE with both automated baselines and human-recorded test cases. The results

demonstrate that our approach achieves superior bug detection capability and test case diversity, while reducing execution cost by a large margin compared to baselines. Moreover, our ablation studies confirm the critical contributions of test case optimization and update-aware prioritization, which significantly enhance regression testing performance.

For future work, we plan to extend our study in the following directions. First, we plan to extend our evaluation to practical game development settings. The current experiments were conducted in controlled research environments with structured tasks and update logs, which may differ from the noisy, heterogeneous, and large-scale conditions of commercial game development. Collaborating with industry partners will allow us to assess the practical utility of our framework in real production pipelines. Second, we aim to further reduce the computational overhead of our method. While the proposed multi-objective optimization significantly improves test case quality, the path extraction and evaluation process can still become expensive in large state–action graphs. A promising direction is to integrate pruning strategies into the optimization stage, so that redundant or low-value paths can be efficiently discarded without sacrificing behavioral diversity. Such hybrid optimization–pruning techniques could greatly enhance the scalability and responsiveness of our approach in resource-constrained real-world workflows. Third, we will explore graph-based incremental maintenance mechanisms to improve long-term adaptability across version updates. Our current strategy focuses on re-execution and removal of obsolete test cases, but this process can become costly over extended development cycles. By explicitly modeling version-to-version transitions as update-aware graphs, we aim to incrementally adapt test suites with minimal recomputation. This approach has the potential to maintain consistency and stability of regression testing pipelines while reducing manual intervention and long-term overhead.

Acknowledgement

This work was conducted during the author's visit to Southwest University. This research was partially supported by JSPS KAKENHI (No. 23K28064, 25K15290) and JST SPRING(No. JPMJSP2128).

Data Accessibility

The complete replication suite for the Overcooked Plus environment is publicly available at our project repository: https://github.com/BlueLinkX/SAGE . This includes the environment's source code, task configurations, embedded bug triggers, and all generated metadata used in our evaluation. For Minecraft, the source code is not public as the experiments were conducted using a custom modification (mod) that relies on Mojang/Microsoft's proprietary APIs. However, the code can be shared upon reasonable request for academic use.

A Appendix I: Example Update Log

Appendix A presents an illustrative example of an update document, showcasing the general format of version descriptions used in our experiments.

Example Update Log: Overcooked Plus (Version 2.0)

Release Date: September 1, 2025
New Features:

- *Cheese Cutting Mechanic:* Cheese must now be chopped before use, introducing a new `chopped_cheese` state.
- *Soup Crafting System:* Added two new recipes, `MeatSoup` and `FishSoup`, with multi-ingredient dependencies (e.g., tomato + onion + meat/fish).
- ...

Feature Changes:

- Unified inheritance between `Fish` and `Meat` classes for consistent cooking logic.
- Adjusted interaction priorities between plates and cooking devices to reduce ambiguity.
- ...

New Tasks:

- *Cheese Pizza:* Involves baking dough, slicing cheese, and assembling with tomato sauce before serving.
- *Combo Meal Challenge:* A multi-step mission combining soup, baked bread, and grilled meat under strict time constraints.
- ...

Bug Fixes:

- Fixed an issue where chopped onions occasionally disappeared when placed on cutting boards.
- Fixed visual glitch where burned pizza retained its uncooked texture.
- ...

...

B Appendix II: Example LLM Prompts

This appendix provides representative examples of the prompts employed in SAGE, illustrating how seed trajectories are generated for RL training and how large language models are guided to extract semantic tags from update logs.

Listing 1 Example JSON prompt for generating a seed trajectory

```
{
  "environment": {
    "name": "Overcooked Plus",
    "game_description": "This is a cooperative cooking simulation game where players must coordinate to prepare meals under time constraints...",
    "basic_rules": "Each ingredient must be processed correctly before serving...",
    "current_obs_description": "Kitchen layout: dough at (1,3), tomatoes at (...), cheese at (...), oven at (...), serving table at (...), ...",
    "available_actions": [
```

```
      "move_up", "move_down", "move_left", "move_right",
      "pickup", "drop", "interact", ...
    ]
  },

  "task": {
    "name": "Make Pizza",
    "task_objective": "Combine processed dough, tomato, and cheese into a pizza and bake
        it to completion...",
    "related_rules": "Dough must be kneaded before use; tomatoes must be chopped; cheese
        must be sliced..."
  },

  "past_solutions": [
    {
      "summary": ...,
      "key_subtasks": ...
    }
  ],

  "instructions": "Generate a new plan different from past_solutions to complete the
      task. Decompose the task into subtasks (e.g., fetch ingredients, chop, assemble,
      bake, serve). For each step, state its goal and the atomic action from
      available_actions. Ensure logical consistency and strategy diversity (e.g.,
      different order, shorter path, or parallel workflow).",

  "output_format": "Return a JSON object containing: 1. summary: overall description of
      the new plan; 2. key_steps: list of main subtasks; 3. steps: an array of {step,
      description, action}."
}
```

Listing 2 Example prompt for extracting game component tags

```
{
  "update log": "Release Date: September 1, 2025\n New Features:....",

  "instructions": "Read the given update log text and identify keywords representing
      affected game components, including but not limited to: items, actions, UI,
      functions, environment, and mechanics. Do not extract version numbers or filler
      words. Each tag should be a lowercase phrase.",

  "output_format": "Return a JSON object with the following fields: 1. tags: a list of
      extracted keyword tags; 2. summary: a one-sentence natural language summary of
      the log entry."
}
```

References

[1] Newzoo: Newzoo's Global Games Market Report 2024 - Free Version. Accessed: 2024-10-29 (2024). https://newzoo.com/resources/trend-reports/newzoos-global-games-market-report-2024-free-version

[2] Wu, Y., Chen, Y., Xie, X., Yu, B., Fan, C., Ma, L.: Regression testing of massively multiplayer online role-playing games. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 692–696 (2020). https://doi.org/10.1109/ICSME46990.2020.00074

[3] Agrawal, H., Horgan, J.R., Krauser, E.W., London, S.A.: Incremental regression testing. In: 1993 Conference on Software Maintenance, pp. 348–357 (1993). https://doi.org/10.1109/ICSM.1993.366927

[4] Gligoric, M., Eloussi, L., Marinov, D.: Practical regression test selection with dynamic file dependencies. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015, pp. 211–222. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2771783.2771784

[5] Netravali, R., Sivaraman, A., Das, S., Goyal, A., Winstein, K., Mickens, J., Balakrishnan, H.: Mahimahi: Accurate Record-and-Replay for HTTP. In: 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 417–429. USENIX Association, Santa Clara, CA (2015). https://www.usenix.org/conference/atc15/technical-session/presentation/netravali

[6] Ostrowski, M., Aroudj, S.: Automated regression testing within video game development. GSTF Journal on Computing (JoC) **3**(2), 10 (2013) https://doi.org/10.7603/s40601-013-0010-4

[7] Gordillo, C., Bergdahl, J., Tollmar, K., Gisslén, L.: Improving playtesting coverage via curiosity driven reinforcement learning agents. In: 2021 IEEE Conference on Games (CoG), pp. 1–8 (2021). https://doi.org/10.1109/CoG52621.2021.9619048

[8] Yu, J., Wu, Y., Xie, X., Le, W., Ma, L., Chen, Y., Hu, J., Zhang, F.: Gamerts: A regression testing framework for video games. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1393–1404 (2023). https://doi.org/10.1109/ICSE48619.2023.00122

[9] Duarte, Y., Canella, H., Durelli, V., Nardi, P., Endo, A.: Exploratory testing for platform video games: strategies and lessons learned. Journal on Interactive Systems **15**, 657–669 (2024) https://doi.org/10.5753/jis.2024.4156

[10] Mingyue, Z., Xiao-Yi, Z., Paolo, A., Fuyuki, I.: An investigation of the behaviours of machine learning agents used in the game of go. 2023 10th International Conference on Dependable Systems and Their Applications (DSA), 734–742 (2023) https://doi.org/10.1109/dsa59317.2023.00105

[11] Politowski, C., Petrillo, F., Guéhéneuc, Y.-G.: A survey of video game testing. IEEE/ACM International Conference on Automation of Software Test (AST), 90–99 (2021) https://doi.org/10.1109/AST52587.2021.00018

[12] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., Postma, E.: Adaptive game ai with dynamic scripting. Machine Learning **63**(3), 217–248 (2006) https://doi.org/10.1007/s10994-006-6205-6

[13] Mioto, V., Petrillo, F.: A mapping of recording-based game test automation tools. In: 2025 IEEE/ACM 9th International Workshop on Games and Software Engineering (GAS), pp. 1–8 (2025). https://doi.org/10.1109/GAS66647.2025.00006

[14] Stahlke, S., Nova, A., Mirza-Babaei, P.: Artificial players in the design process: Developing an automated testing tool for game level and world design. In: Proceedings of the Annual Symposium on Computer-Human Interaction in Play. CHI PLAY '20, pp. 267–280. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3410404.3414249 . https://doi.org/10.1145/3410404.3414249

[15] Iftikhar, S., Iqbal, M.Z., Khan, M.U., Mahmood, W.: An automated model based testing approach for platform games. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 426–435 (2015). https://doi.org/10.1109/MODELS.2015.7338274

[16] Stahlke, S.., Nova, A., Mirza-Babaei, P.: Artificial playfulness: A tool for automated agent-based playtesting. In: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems. CHI EA '19, pp. 1–6. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3290607.3313039

[17] Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., Liu, Y., Shen, R., Chen, Y., Fan, C.: Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 772–784 (2019). https://doi.org/10.1109/ASE.2019.00077

[18] Guerrero-Romero, C., Lucas, S.M., Perez-Liebana, D.: Using a team of general ai algorithms to assist game design and testing. In: 2018 IEEE Conference on Computational Intelligence and Games (CIG), pp. 1–8 (2018). https://doi.org/10.1109/CIG.2018.8490417

[19] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q.: Software testing with large language models: Survey, landscape, and vision. IEEE Transactions on Software Engineering **50**(4), 911–936 (2024) https://doi.org/10.1109/TSE.2024.3368208

[20] Tufano, M., et al.: Unit test case generation with transformers and focal context. arXiv preprint arXiv:2009.05617 (2020)

[21] Alagarsamy, S., Tantithamthavorn, C., Aleti, A.: A3test: Assertion-augmented automated test case generation. Information and Software Technology **176**, 107565 (2024) https://doi.org/10.1016/j.infsof.2024.107565

[22] Xie, Y., et al.: Chatunitest: A chatgpt-based automated unit test generation tool. arXiv preprint arXiv:2305.04764 (2023)

[23] Dakhel, A.M., Nikanjam, A., Majdinasab, V., Khomh, F., Desmarais, M.C.: Effective test

generation using pre-trained large language models and mutation testing. Information and Software Technology **171**, 107468 (2024) https://doi.org/10.1016/j.infsof.2024.107468

[24] Mastropaolo, A., Cooper, N., Palacio, D.N., Scalabrino, S., Poshyvanyk, D., Oliveto, R., Bavota, G.: Using transfer learning for code-related tasks. IEEE Transactions on Software Engineering **49**(4), 1580–1598 (2022) https://doi.org/10.1109/TSE.2022.3183297

[25] Tufano, M., Drain, D., Svyatkovskiy, A., Sundaresan, N.: Generating accurate assert statements for unit test cases using pretrained transformers. In: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test. AST '22, pp. 54–64. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524481.3527220 . https://doi.org/10.1145/3524481.3527220

[26] Nashid, N., Sintaha, M., Mesbah, A.: Retrieval-based prompt selection for code-related few-shot learning. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 2450–2462 (2023). https://doi.org/10.1109/ICSE48619.2023.00205

[27] Liu, Z., Chen, C., Wang, J., Che, X., Huang, Y., Hu, J., Wang, Q.: Fill in the blank: Context-aware automated text input generation for mobile gui testing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1355–1367 (2023). https://doi.org/10.1109/ICSE48619.2023.00119

[28] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., Wang, Q.: Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3597503.3639180

[29] Deng, Z., et al.: Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt. arXiv preprint arXiv:2304.02014 (2023)

[30] Deng, Y., Xia, C.S., Peng, H., Yang, C., Zhang, L.: Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 423–435 (2023). https://doi.org/10.1145/3597926.3598067

[31] Zhang, T., Irsan, I.C., Thung, F., Han, D., Lo, D., Jiang, L.: itiger: An automatic issue title generation tool. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1637–1641 (2022). https://doi.org/10.1145/3540250.3558934

[32] Bui, N.D.Q., Wang, Y., Hoi, S.C.H.: Detect–localize–repair: A unified framework for learning to debug with codet5, 812–823 (2022) https://doi.org/10.18653/v1/2022.findings-emnlp.57

[33] Chen, S., et al.: Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128 (2023)

[34] Feng, S., Chen, C.: Prompting is all you need: Automated android bug replay with large language models. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 1–13 (2024). https://doi.org/10.1145/3597503.3608137

[35] Lajkó, M., Csuvik, V., Vidács, L.: Towards javascript program repair with generative pre-trained transformer (gpt-2). In: Proceedings of the Third International Workshop on Automated Program Repair. APR '22, pp. 61–68. Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3524459.3527350

[36] Wang, W., Wang, Y., Joty, S., Hoi, S.C.H.: Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 146–158 (2023). https://doi.org/10.1145/3611643.3616256

[37] iXie Gaming: A Comprehensive Review of Game Test Automation Tools. Accessed: 2025-10-14 (2024). https://www.ixiegaming.com/blog/comprehensive-review-game-test-automation-tools/

[38] De Souza, L.S., Prudêncio, R.B.C., Barros, F.d.A.: A hybrid binary multi-objective particle swarm optimization with local search for test case selection. In: 2014 Brazilian Conference on Intelligent Systems (BRACIS), pp. 414–419 (2014). https://doi.org/10.1109/BRACIS.2014.80

[39] Mondal, D., Hemmati, H., Durocher, S.: Exploring test suite diversification and code coverage in multi-objective test case selection. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–10 (2015). https://doi.org/10.1109/ICST.2015.7102588

[40] Souza, L.S.d., Miranda, P.B.C.d., Prudencio, R.B.C., Barros, F.d.A.: A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort. In: 2011 IEEE 23rd International Conference on Tools with Artificial Intelligence, pp. 245–252 (2011). https://doi.org/10.1109/ICTAI.2011.45

[41] Cai, J., Li, J., Li, N., Zhang, M., Yang, R., Tei, K.: Overcooked plus: A comprehensive cooking scenario testbed for enhancing the evaluation of autonomous planning algorithms. In: 2024 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 146–151 (2024). https://doi.org/10.1109/ACSOS-C63493.2024.00046

[42] Ghost Town Games Ltd.: Overcooked! Accessed: 2024-08-26 (2024). https://www.

team17.com/games/overcooked/ Accessed 2024-08-26

[43] Mojang Studios: Minecraft. https://www.minecraft.net. Sandbox video game developed by Mojang Studios (2009)

[44] Hu, J., Zhang, M., Liu, B., Wu, Y., Chen, Y.: A language-guided acceleration method for smoke testing of game quests. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 7–12 (2024). https://doi.org/10.1109/ISSREW63542.2024.00039

[45] Li, Z., Wu, Y., Ma, L., Xie, X., Chen, Y., Fan, C.: Gbgallery: A benchmark and framework for game testing. Empirical Software Engineering **27**(6), 140 (2022) https://doi.org/10.1007/s10664-022-10158-x

[46] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal Policy Optimization Algorithms (2017). https://arxiv.org/abs/1707.06347

[47] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., Dormann, N.: Stable-baselines3: Reliable reinforcement learning implementations. Journal of machine learning research **22**(268), 1–8 (2021)

[48] Valve Corporation: Steam — The Ultimate Online Game Platform. Accessed on October 30, 2025 (2025). https://store.steampowered.com/