RPG DREAMER: TOWARDS AGENT-DRIVEN GENERATION OF LONG-HORIZON INTERACTIVE WORLDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Games, as a special category of software, create interactive worlds governed by their own specific rules. Given the recent successes of coding agents, increasing attention is devoted to automatic game generation. We observe that existing methods fail to create long-horizon games, as they entangle game mechanics with content and lack effective playtesting capabilities. In this work, we present RPG Dreamer, a framework capable of generating games with hours-long coherent playthroughs. We use role-playing games (RPGs) to exemplify the long-horizon generation capabilities of the framework, as RPGs feature branching narratives, non-linear progressions, and extended playtime. RPG Dreamer consists of an Agent-Oriented Game Engine (AOGE) and a Reviewer-centric Generation Loop (RGL). AOGE separates mechanics from content and exposes an agent interface with uniform observation and semantic action, enhancing both the reliability and coverage of automated gameplay. RGL is a closed-loop pipeline that enables agents to iteratively generate, review, and evolve games through a multi-reviewer system, including a team of gameplay reviewers who conduct playtesting to expose latent bugs. Due to the absence of benchmarks for evaluating hour-scale game generation, we further propose RPG-Bench, a benchmark of 150 evaluation instances spanning diverse themes, whose metrics are grounded in dynamic gameplay trajectories. Results on RPG-Bench demonstrate that RPG Dreamer improves the scalability of state-of-the-art coding agents on long-horizon games.

1 INTRODUCTION

With recent advances in foundation multimodal large language models (MLLMs) and harness engineering (Qwen Team, 2026; Kimi Team, 2026; OpenAI, 2026; Anthropic, 2026), creating games via coding agents (Jiang et al., 2026b; Huang et al., 2026b) is drawing increasing research attention. As a special category of software, games create virtual worlds governed by their own rules. Their more diverse interactions, richer multimodal assets, and more subjective evaluation rubrics pose substantial challenges to automatic generation (Barr et al., 2007; Bellotti, 2007).

Automatic game generation is rooted in decades-long research on procedural content generation (PCG). Early methods typically rely on domain-specific representations, handcrafted generators, or explicitly defined search objectives, making them effective but tightly scoped to particular game types (Togelius et al., 2011; Hendrikx et al., 2013). Later work increases the flexibility and scope of PCG by generating diverse, controllable levels and by composing stories, narratives, and tile-based environments into coherent game worlds (Todd et al., 2023; Sudhakaran et al., 2023; Nasir et al., 2024). With recent advances in coding agents, automatic game generation expands beyond individual content artifacts toward complete playable games, with coding agents jointly constructing source code, gameplay logic, scenes, and assets from natural language requirements (Jiang et al., 2026b; Yin et al., 2026; Huang et al., 2026b). These methods improve game generation quality through engine-specific workflows, domain-specialized models, and GUI-driven refinement. However, they focus on creating games with short gameplay progression and simple logic. We put forward a key question: *What hinders them from creating long-horizon, complex, and functional games?*

Key observations. To find the reason, we conduct exploratory studies. We first quantitatively demonstrate that existing methods exhibit a rapid drop in generation quality as games become more

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

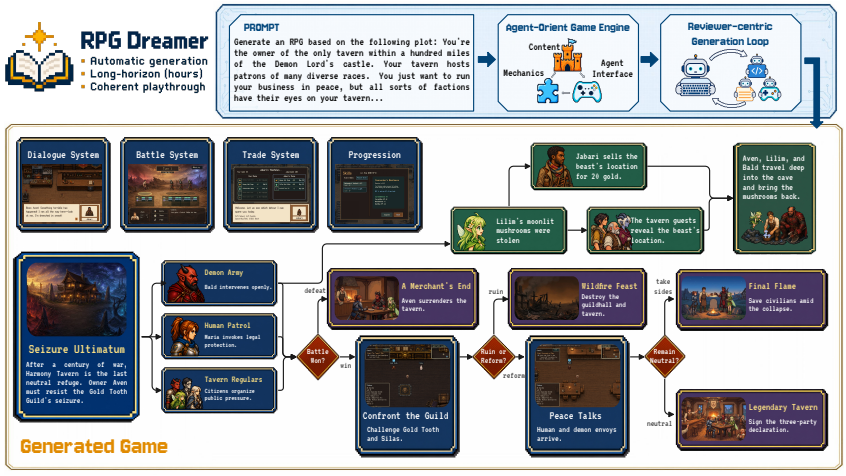


Figure 1: Capability overview of RPG DREAMER. Given a prompt, RPG Dreamer combines an Agent-Oriented Game Engine with the Reviewer-centric Generation Loop to generate a complete game. The generated game integrates dialogue, battle, and trading systems across a branching narrative with multiple endings, delivering a long-horizon and coherent gameplay experience.

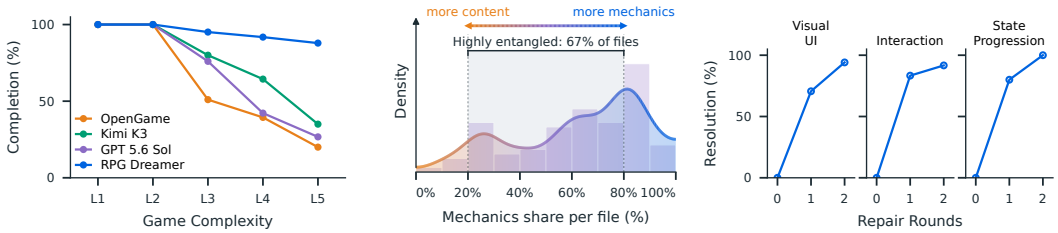
complex (figure 2a). Next, according to game design theory (Salen & Zimmerman, 2003), games can be decomposed into two parts: *mechanics*, which define reusable and underlying rules, and *content*, which instantiates the rules with specific assets, levels, and other artifacts. However, our analysis of generation trajectories and output projects (figure 2b) reveals that existing methods frequently entangle mechanics with content. This entanglement blurs the boundary between game mechanics and content during generation, making bugs hard to detect and complicating maintenance. In addition, we manually collect the bugs in the generated complex games and analyze why these bugs remain unresolved. We find that coding agents are largely capable of fixing bugs once these bugs are brought to their attention (figure 2c), which demonstrates that coding agents lack effective approaches to detect the existence of bugs during generation. Therefore, we argue that the core bottlenecks lie in *the entanglement between reusable mechanics and game-specific content, as well as the insufficiency of current approaches for discovering bugs*.

Our solution. In this work, we propose a new framework that explicitly separates mechanics creation and content generation into two stages and introduce a gameplay-based playtest method. We demonstrate this framework on role-playing games (RPGs), a genre whose branching stories, nonlinear progressions, and long gameplay cycles make it a representative testbed for the existing challenges. We name our framework **RPG Dreamer**, which consists of two core modules: an *Agent-Oriented Game Engine* (AOGE) and a *Reviewer-centric Generation Loop* (RGL). AOGE consists of three components: 1) The mechanics component defines reusable gameplay rules and interaction logic, such as turn-based combat resolution and branching dialogue trees. 2) The content component instantiates these generic mechanics with specific assets, levels, narratives, and quests. 3) The agentic interface component provides uniform observation and action interfaces for agents to perceive game states and interact with games. Accordingly, game generation proceeds in two stages: the first stage constructs reusable mechanics and the agentic interface, while the second stage generates game-specific content. For the second stage (i.e., the content-generation stage), we propose RGL, a closed-loop pipeline in which agents iteratively generate, review, and evolve game content. During review, in addition to conventional static checks, we employ a team of gameplay reviewers to conduct playtests by actually playing the games. With the agent interface, gameplay reviewers can seamlessly play long-horizon games and provide diagnostic feedback, which helps locate latent bugs manifesting in the late stages of gameplay.

To rigorously evaluate our framework, a benchmark needs the ability to assess the generation of complete long-horizon games, yet existing benchmarks are limited to simple games (Jiang et al., 2026b), isolated development subtasks (Chi et al., 2026; La et al., 2026), or narrow playtests (Yin et al., 2026; Huang et al., 2026b; Luo et al., 2026). We therefore introduce **RPG-Bench**, a benchmark comprising 150 evaluation instances for generating hour-scale RPGs across diverse genres.

Table 1: Comparisons between RPG-Bench and existing benchmarks. Complete game generation includes both code and assets.

Benchmark	Evaluation Instances	Complete Game Generation	Evaluated by Gameplay	Long-Horizon
GameDevBench (Chi et al., 2026)	132	×	×	×
GameEngineBench (La et al., 2026)	110	×	×	×
OpenGame-Bench (Jiang et al., 2026b)	150	✓	×	×
PlayGen-20 (Yin et al., 2026)	20	×	limited actions	×
PlaytestArena (Huang et al., 2026b)	200	✓	low-level control	×
GameCraft-Bench (Luo et al., 2026)	140	×	predefined routes	×
GBQA (Jiang et al., 2026a)	30	×	w/o rendered UI	×
RPG-Bench (ours)	150	✓	**dynamic & diverse playtesting**	✓



(a) Generation quality decreases as complexity increases.

(b) Mechanics and content are entangled.

(c) Once reported by humans, bugs can be fixed by the agent.

Figure 2: Exploratory studies of the bottlenecks in current game-generation methods with details described in section 2.2.

Unlike other benchmarks, RPG-Bench grounds its metrics in dynamic gameplay trajectories, covering both normal multi-branch playthroughs and unconventional actions, evaluating generated games in terms of requirement fidelity, content richness, visual and interaction quality, and playability.

Our main contributions are summarized as follows:

- We propose **RPG Dreamer**, a framework that automatically generates complete games including game logic, content, and multimodal assets, scaling the capabilities of existing coding agents to coherent gameplay over hour-scale playthroughs.
- We design the two core modules of RPG Dreamer: **Agent-Oriented Game Engine (AOGE)** and **Reviewer-centric Generation Loop (RGL)**. AOGE decouples reusable game mechanics from game-specific content and provides unified observation and action interfaces for agents. RGL is a closed-loop generation pipeline that enables iterative evolution of generated game content through a multi-reviewer system, including a team of gameplay agents that target latent bugs via playtesting.
- We present **RPG-Bench**, a benchmark comprising 150 evaluation instances for generating hour-scale RPGs across diverse genres and themes, with metrics grounded in dynamic gameplay trajectories that assess requirement fidelity, content richness, visual and interaction quality, and playability.

2 Background and Motivation

2.1 Problem Definition

In this paper, we focus on the task of automatically generating games with complete logic and assets. This task requires a coding agent $\mathcal{F}$ to turn a user requirement u into a complete game

$$G = \mathcal{F}(u). \tag{1}$$

The resulting game is not merely software but an interactive world with its own rules and progression in which the player acts and observes. At each step t, the player observes $o_t = \Omega_G(s_t)$, selects an

action a_t from the actions $\mathcal{U}_G(s_t)$ afforded in state s_t, and the world advances according to

$$s_{t+1} = T_G(s_t, a_t), \tag{2}$$

where T_G is the transition function implemented by the game's logic. For example, if s_t records that the player stands before a locked gate and carries its key, then applying the action $a_t =$ "unlock the gate" produces a new state s_{t+1} in which the gate is open and the area beyond it is accessible. A playthrough of G therefore induces a trajectory $\tau = (s_0, a_0, \ldots, s_H)$, where H is the number of steps at which the playthrough terminates. Playing G induces a distribution $\mathcal{D}_G$ over such trajectories. Given an evaluation function $V(\tau; u)$ that scores a trajectory along multiple dimensions, including its fidelity to the requirement u, visual quality, and overall playability, the goal is to maximize the expected score over the trajectories the game induces,

$$G^\star = \arg\max_G \ \mathbb{E}_{\tau \sim \mathcal{D}_G}\big[V(\tau; u)\big]. \tag{3}$$

2.2 Exploratory Study

Generation quality decreases as complexity increases. First, to identify the bottlenecks in generating long games, we construct five RPG generation tasks of increasing complexity, denoted L1–L5: L1 requests only a minimal single-scene game with a linear storyline, whereas L5 demands a complex game with multiple scenes, branching plots, and multiple endings (See section E.1.2 for details). At every level, three representative baselines (GPT-5.6 Sol (OpenAI, 2026), Kimi K3 (Kimi Team, 2026), and OpenGame (Jiang et al., 2026b)) each generate five games from the same five stories. For each generated game we then extract its complete branching structure, construct playthrough routes that jointly cover all available choices, and manually play each route to determine whether it successfully terminates at an ending and compute the corresponding completion rate (See section E.1 for details). The results in figure 2a show that the completion rate declines rapidly as complexity grows. At L5, less than half of the routes can be completed.

Mechanics and content are entangled in process and output. To localize the source of this degradation, we inspect the generation trajectories of the five L5 games. We find that coding agents invoke file-edit tools to produce mechanics-related and content-related outputs simultaneously in 68.4% of cases on average (mechanics and content follow the distinction of Salen & Zimmerman (2003)), excluding third-party libraries, test files, and temporary files. To examine whether this mixing also persists in the delivered artifacts, we then analyze the output projects. For each file, multiple coding agents independently estimate the shares of its code lines belonging to mechanics and to content, and we average their estimates (details in section E.2). Figure 2b shows that 67% of files fall in the highly entangled 20–80% range, rather than concentrating in dedicated modules. This entanglement blurs the mechanics–content boundary, hiding bugs from detection and complicating maintenance.

Once reported by humans, bugs can be fixed by the agent. We further analyze the unresolved bugs in the generated complex games that underlie the degradation, to determine whether the agents cannot repair them or simply never discover them. To distinguish the two cases, we collect confirmed bugs from human playthroughs of the five L5 games generated by each of the three methods and report them back to the generation agents. We manually group the confirmed bugs into three categories, visual/UI, interaction, and state-progression bugs (details in section E.3). As shown in figure 2c, Round 0 represents the initial generated games delivered after the agents' own self-review, where all collected bugs remain unresolved. We then plot the cumulative repair rate over two rounds of repair. We find that once the agent observes the bug, it effectively repairs most within one round and nearly all within two, indicating that the bottleneck is bug discovery rather than repair.

Implications. According to Salen & Zimmerman (2003), a game's formal rules and its representational content can be treated as separate concerns. However, agents in current generation pipelines often entangle mechanics with content in both the generation process and output files, which blurs the boundary between game mechanics and content, making bugs hard to detect and complicating maintenance. Moreover, we find that many bugs remain not because they exceed the agents' repair capabilities but because agents lack effective approaches to observe them. These findings motivate both an explicit isolation of reusable mechanics and game-specific content and an effective playtesting method that can expose bugs in complex games with long interaction horizons.

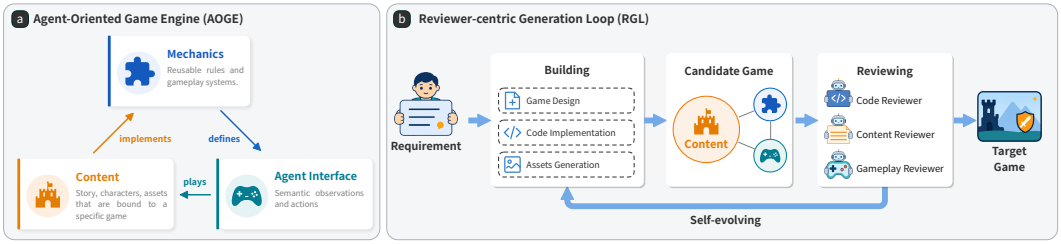


Figure 3: Overview of RPG DREAMER. **(a)** AOGE consists of reusable mechanics, content slots, and a mechanics-grounded Agent Interface. **(b)** RGL instantiates the content slots by producing the game design, code implementation, and assets, assembles the resulting content with the mechanics and Agent Interface into a candidate game, and evaluates it with code, content, and gameplay reviewers. Review feedback returns to the building process, forming a self-evolving loop that iteratively refines the candidate into the target game.

3 METHOD

According to the findings above on mechanics–content entanglement and inadequate testing for long-horizon gameplay, we introduce RPG Dreamer, a framework for generating complex, long-horizon RPGs from natural-language requirements. As illustrated in figure 3, it creates games in two stages. The first stage manually constructs an Agent-Oriented Game Engine (AOGE), comprising reusable mechanics, content slots, and a mechanics-grounded Agent Interface through which agents observe game state and issue semantic actions. With the mechanics and interface fixed, the second stage employs the Reviewer-centric Generation Loop (RGL) to automatically populate game-specific content. As more capable models depend less on rigid procedural constraints, this stage focuses on multidimensional review through both static checks and dynamic full-path playtesting.

3.1 AGENT-ORIENTED GAME ENGINE

To explicitly separate game mechanics from game content and enable more effective playtesting, we propose the Agent-Oriented Game Engine (AOGE) for generating high-quality long-horizon games. Built on top of a conventional game engine, AOGE delivers an organizational paradigm and components tailored for game generation with coding agents. The first stage constructs three components for a target game type: Mechanics $\mathcal{M}$, Content Slots $\overline{\mathcal{C}}$, and the Agent Interface $\mathcal{I}$, which together form an AOGE:

$$\text{AOGE} = (\mathcal{M}, \overline{\mathcal{C}}, \mathcal{I}). \tag{4}$$

3.1.1 MECHANICS AND CONTENT SLOTS

We define Mechanics $\mathcal{M}$ as the gameplay rules and systems reusable across a family of games, and Content $\mathcal{C}$ as the story, characters, scenes, and assets tied to a specific game. Content Slots $\overline{\mathcal{C}}$ specify the structure, admissible fields, and constraints of $\mathcal{C}$ without fixing its values, so content freedom can range from filling in attribute values within a fixed schema to defining relations among content elements. Section F.1 provides two representative examples of this spectrum. When mechanics are runnable gameplay logic rather than static templates, content can define composition and control flow, not merely fill in attributes.

The boundary between mechanics and content is a design choice that depends on what the framework predefines and what remains open to generation. AOGE makes it configurable per game type, following the principle that mechanics guarantee integrity. By predefining functionality as reusable, testable gameplay logic, the framework grants content as much freedom as the game requires without compromising reliability. This principle is realized when constructing an AOGE by building $\mathcal{M}$ and reserving $\overline{\mathcal{C}}$ for the target game type. The resulting mechanics–content separation makes the integrity guarantee concrete by reducing quality degradation as complexity grows.

3.1.2 AGENT INTERFACE

Having a coding agent drive a game with raw keyboard and mouse inputs is difficult, and existing automated gameplay methods (Zhang et al., 2025; Zhao & Tang, 2025) are tied to a single game type and generalize unsatisfactorily. Inspired by PrismarineJS (2013); Jiang et al. (2026a), we propose an Agent Interface $\mathcal{I}$. The agent interface abstracts the action space to the intent level, letting the agent operate the game by issuing intent-level instructions such as "talk to the blacksmith" or "move to the forest scene". The interface remains strictly grounded in real player operations, so the agent cannot pass the game by injecting variables or modifying saves.

The Agent Interface operates only on the game's action space, which is determined solely by the game mechanics and is independent of the content. It is therefore constructed together with the mechanics in the first stage, before any game-specific content exists. This ordering also avoids the shortcut of jointly generating a game with its testing interface, where the generator tends to expose endpoints only for functionality it has already verified.

3.2 REVIEWER-CENTRIC GENERATION LOOP

Let u denote the requirements for the content of a specific game. Given u and an AOGE $(\mathcal{M}, \overline{\mathcal{C}}, \mathcal{I})$, RGL instantiates the content slots $\overline{\mathcal{C}}$ as game-specific content $\mathcal{C}$ by iteratively building a candidate, exposing its defects through review, and repairing them until the requirements are satisfied. The resulting game is $G = (\mathcal{M}, \mathcal{C}, \mathcal{I})$, where $\mathcal{C}$ conforms to the structure and constraints specified by $\overline{\mathcal{C}}$. Algorithm 1 summarizes the complete procedure, which serves as the sole formal specification of the loop.

Algorithm 1 Reviewer-Centric Generation Loop

Require: Requirements u, AOGE $(\mathcal{M}, \overline{\mathcal{C}}, \mathcal{I})$, maximum rounds K
Ensure: Delivered game G

$\mathcal{C}_0 \leftarrow \varnothing$; $Q_0 \leftarrow \varnothing$
for $k = 1, \ldots, K$ **do**
  $\mathcal{C}_k \leftarrow \textsc{Build}(u, \overline{\mathcal{C}}, \mathcal{C}_{k-1}, Q_{k-1})$
  $G_k \leftarrow (\mathcal{M}, \mathcal{C}_k, \mathcal{I})$
  $Q_{k,\text{code}} \leftarrow \textsc{Reviewer}_{\text{code}}(G_k)$
  $Q_{k,\text{content}} \leftarrow \textsc{Reviewer}_{\text{content}}(G_k, u)$
  $Q_{k,\text{play}} \leftarrow \textsc{Reviewer}_{\text{play}}(G_k)$
  $Q_k \leftarrow Q_{k,\text{code}} \cup Q_{k,\text{content}} \cup Q_{k,\text{play}}$
  if $Q_k = \varnothing$ **then**
    return G_k
  end if
end for
return G_K

3.2.1 GAME BUILDING

Construction follows three core procedures: drafting the design document (GDD), writing source code, and generating game assets. As the game mechanics $\mathcal{M}$, content-slot specification $\overline{\mathcal{C}}$, and agent interface $\mathcal{I}$ are inherited from the AOGE and fixed across the entire iterative loop, our generation module exclusively produces the game-specific content $\mathcal{C}$. At the initial iteration, both $\mathcal{C}_0$ and Q_0 are empty. In subsequent iterations, the builder iteratively refines $\mathcal{C}_k$ to address the issues summarized in Q_{k-1}. Notably, the GDD serves as both the foundational design blueprint and the evaluation contract for auditing all produced game artifacts throughout the refinement process. The builder $\textsc{Build}(u, \overline{\mathcal{C}}, \mathcal{C}_{k-1}, Q_{k-1})$ takes the game-specific content requirements u, content-slot specification $\overline{\mathcal{C}}$, content from the previous iteration $\mathcal{C}_{k-1}$, and review feedback Q_{k-1} as inputs, and outputs updated game content $\mathcal{C}_k$.

3.2.2 GAME REVIEWING

As the feedback step in the iterative loop, game reviewing triggers immediately after the builder generates a new game candidate, judging whether the candidate is deliverable. If not, it produces corrective guidance for subsequent iterations. This step relies on three complementary reviewers:

- **Code reviewer** verifies the local correctness, compilation compatibility, and build validity of the generated candidate via deterministic automated scripts. The inspection covers installation procedures, structural schemas, referential integrity, mechanic unit tests, progression reachability, and save/load equivalence.
- **Content reviewer** identifies semantic misalignment between the generated game content and the intended design plan or game-specific content requirements u. It employs dedicated subagents to examine the GDD, content graph, and asset bindings.

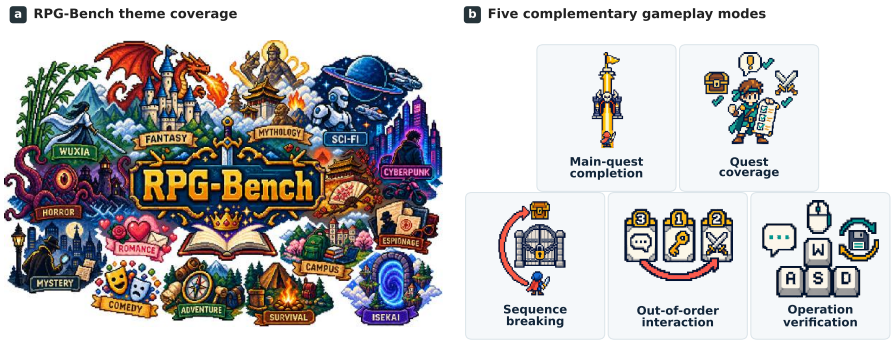


Figure 4: Overview of the theme coverage of RPG-Bench and the gameplay modes for agent-based evaluation. RPG-Bench contains 150 evaluation instances, and evaluators are required to play the game under five complementary modes to examine game quality.

- **Gameplay reviewer** assesses the overall functional integrity and playability of the resulting game flow. A playtest agent first samples diverse execution routes from the game source code, where each route $\rho = (a_0, a_1, \ldots, a_{n-1})$ is an action sequence of length n accessible via the agent interface $\mathcal{I}$. For example, in RPG scenarios, such routes cover all story branches. The agent executes the routes in parallel in a headless browser through $\mathcal{I}$ and caches shared behavioral prefixes to improve efficiency.

All three reviewers produce structured issue records $q = (\ell, \sigma, p)$, where ℓ denotes the violated requirement, σ indicates the corresponding severity, and p is a minimal execution prefix for reproduction. Individual issue records are aggregated to form the final feedback set Q for iterative refinement. The candidate is delivered once all reviewer-reported issues are resolved or the maximum number of refinement rounds K is reached.

4 RPG-BENCH

To evaluate the framework, we introduce RPG-Bench, a benchmark for the automatic generation of playable, long-horizon RPGs. Existing benchmarks primarily target simple games or isolated game-development subtasks, without evaluating complete long-horizon game generation. RPGs, as a genre integrating long-running main quests with branching side content, exhibit state-dependent out-of-order interactions and persistent progression across interdependent systems, making them well-suited for evaluating complex, long-horizon game generation.

4.1 BENCHMARK CONSTRUCTION

RPG-Bench comprises 150 evaluation instances built from user requirements that span diverse game lengths, genres, adaptation-based and original designs, and levels of specification detail. Requirements are authored by both game-industry practitioners and outside contributors, and each is manually reviewed by senior game designers to guarantee that it is logically self-consistent, unambiguous, and technically achievable as a self-contained 2D Top-Down executable game. The prompt template used to instantiate a game-generation request from each requirement is provided in section H.1.

Unlike existing benchmarks that score generated games by static inspection or short scripted probes, RPG-Bench evaluates long-horizon games through agent-based gameplay, in which an evaluation agent plays each game through the agent interface in five complementary modes (figure 4): **main-quest completion**, **quest coverage**, **sequence breaking**, **out-of-order interaction**, and **operation verification**. Main-quest completion follows the storyline to check quest completability, ending reachability, and gate ordering, while quest coverage verifies that all main and side content is finishable, consistent, and rewarded. Sequence breaking reaches late content early to test progression guards, while out-of-order interaction reorders legal actions to test persistent-state coherence. Operation verification exercises every advertised action and tests save/load persistence. Each mode yields multiple playthrough trajectories rather than a single run, and a VLM judge scores the recorded

observations along four evaluation dimensions. The detailed specification and prompts of all five modes appear in section G.1.1.

4.2 METRICS

To mitigate the self-preference bias existing in LLM-as-judge paradigms (Wataoka et al., 2024), we adopt the following pipeline: a strong coding agent extracts all information required for evaluation into a single file, and a separate VLM judge subsequently assigns scores solely based on this file. Anonymizing outputs in a shared evidence schema and scoring only this evidence reduces the authorship and stylistic cues through which self-preference can affect evaluation. The recorded trajectories are scored along four dimensions:

- **Requirement fidelity (RF).** Measures whether the game implements the narrative, progression, and setting stated in the user requirement.
- **Content richness (CR).** Assesses the engagement and logical coherence of the game narrative. A coding agent first extracts a summary of the story and quests from the source code, which a VLM judge then scores.
- **Visual and interaction quality (VI).** Validates visual aesthetics and UI functionality from screenshots sampled along the gameplay trajectories, covering both art consistency and interface readability.
- **Playability (PL).** Quantifies gameplay experience per mode: main-quest and quest coverage use completion rates, sequence breaking and out-of-order interaction score detected logical issues, and operation verification uses the share of working modules.

Detailed scoring rules, checklists, and judge prompts are provided in section G.1. Each score is normalized to a 0–100 scale. Since most coding agents inherently guarantee implementation consistency regardless of the actual delivered quality during game generation, RF is assigned a lower weight when computing the overall aggregated score:

$$S_{\text{overall}} = 0.1S_{\text{RF}} + 0.3S_{\text{CR}} + 0.3S_{\text{VI}} + 0.3S_{\text{PL}}. \tag{5}$$

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Baselines. We compare RPG Dreamer against a broad set of state-of-the-art foundation models, including Claude Opus 4.8 (Anthropic, 2026), Qwen3.8-Max (Qwen Team, 2026), Kimi K3 (Kimi Team, 2026), and GPT-5.6 Sol (OpenAI, 2026). Each model is paired with the harness officially recommended by its provider, which runs its own build–test–fix iterations under its own control (section H.1).

Environment and Configuration. For each generation system, we use the maximum context length supported by its underlying model. All baseline agents have access to tools for web search, local file editing, and code execution. To control for asset-generation capabilities, we also equip every baseline with the same asset-generation skills used in RGL, including skills for producing backgrounds, animations, and other game assets. We set the maximum number of RGL build–review rounds to $K = 3$ by default. To balance comprehensive evaluation with experimental efficiency, we use two evaluation sets: RPG-Bench-Full contains all 150 instances for assessing overall performance, whereas RPG-Bench-Val contains a fixed subset of 30 instances for rapid validation and ablation experiments.

5.2 MAIN RESULTS

To evaluate end-to-end RPG generation quality, we compare RPG Dreamer with existing methods and foundation models with harnesses on RPG-Bench. As shown in table 2, existing methods generally fail to achieve satisfactory results in long-horizon game creation. By contrast, RPG Dreamer surpasses the prior state-of-the-art by 29.6% and achieves state-of-the-art overall scores. Moreover,

Table 2: Comparisons between RPG Dreamer and other methods on RPG-Bench-Full.

Harness	Model	RF	CR	VI	PL	Overall
Automatic evaluation						
OpenGame	Opus 4.8	78.3	31.6	41.4	46.0	43.5
Qwen-code	Qwen3.8-Max	84.0	32.2	55.0	29.6	43.4
Kimi-code	Kimi K3	91.7	48.6	67.9	42.3	56.8
Codex	GPT 5.6 Sol	92.3	57.1	55.9	63.5	62.2
RPG Dreamer	GPT 5.6 Sol	**92.7**	**77.1**	**73.2**	**87.4**	**80.6**
Human evaluation						
Codex	GPT 5.6 Sol	44.0	32.1	31.5	34.6	33.9
RPG Dreamer	GPT 5.6 Sol	**69.7**	**61.3**	**70.5**	**62.0**	**65.1**

the performance gap is further widened under human evaluation, with RPG Dreamer outperforming the previous best method by 92.0%. Our user study further shows that from a human perspective, gameplay completeness is critical to game quality. Overly short or incomplete gameplay lowers scores on all dimensions because players never reach the implemented content, and most approaches produce high-quality openings but struggle with subsequent gameplay (comparisons in section D.1).

5.3 Ablations

Table 3: Module ablations on RPG-Bench-Val.

Method	RF	CR	VI	PL	Overall
Baseline	88.0	57.9	56.2	63.8	62.2
Baseline + AOGE	88.0	71.7	61.7	64.4	68.1
Baseline + RGL	92.0	49.6	75.5	85.6	72.4
RPG Dreamer	92.0	75.4	72.6	86.5	79.6

Contribution of modules. To isolate the contributions of AOGE and RGL, we add each module separately to a baseline (Codex GPT 5.6 Sol). As shown in table 3, AOGE substantially improves content richness and visual and interaction quality, since decoupling mechanics from content lets the two stages specialize without cross-task interference. It also moderately improves playability, indicating that this decoupling alone mitigates bugs even without playtests. RGL brings its largest gains in playability together with higher visual and interaction quality, as its playtesting reviewers expose visual anomalies and erroneous gameplay trajectories for repair within the closed loop. Notably, RGL alone slightly reduces content richness, as repair rounds tend to prune problematic content without the decoupling provided by AOGE. The two modules combined yield the best overall score, showing that they provide complementary benefits.

Scaling Performance under Increasing Complexity. We evaluate route completion rates in generated games across progressively increasing complexity levels. As shown in Figure 2a, the completion rates of the other methods decline by an average of 72.8% from L1 to L5, whereas RPG Dreamer exhibits only a 12.1% decrease. This result indicates that RPG Dreamer scales more reliably to longer-horizon game generation. Details are provided in section E.1.

We provide further experiments in the appendix (section C), including validation of the automatic evaluation, evidence for the scale of generated games, fine-grained analysis of individual metrics, and verification of the Agent Interface reliability.

6 Conclusion

We present RPG Dreamer, a framework for long-horizon RPG generation that addresses mechanics–content entanglement and inadequate bug discovery. AOGE separates reusable mechanics from game-specific content and exposes a mechanics-grounded Agent Interface, while RGL iteratively builds and reviews content through code, content, and gameplay feedback. RPG-Bench evaluates complete games from dynamic gameplay trajectories spanning requirement fidelity, content richness, visual and interaction quality, and playability. Experiments show that this separation improves generated content and presentation, gameplay review exposes failures that static checks miss, and the two modules are complementary. These results support executable, feedback-driven generation as a promising direction for interactive worlds. Future work should extend the framework beyond its current web-game setting, reduce generation and evaluation costs, and improve assessment of fine-grained player experience.

AI USE STATEMENT

We use generative AI tools to edit the manuscript language for readability, to create or modify scientific figures and illustrative visual assets, and to assist with implementing our methods. All AI-suggested language edits are reviewed and revised by the authors, every figure is verified against the underlying experimental data, results, and intended content, and all AI-assisted code is reviewed, tested, and verified by the authors before use. We do not use generative AI tools to design the research, formulate hypotheses or mathematical claims, write proofs, generate datasets, clean or reformat data, conduct qualitative analysis, interpret results. The authors take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

We provide the full details needed to reproduce the reported results. For RPG Dreamer, we describe the complete architecture and review workflow, including the Agent-Oriented Game Engine interface and the prompts of the generation and reviewer agents. For RPG-Bench, we provide the requirement template and instance-construction procedure, the scoring rubrics, and the evaluation prompts and procedures for all five gameplay modes; the full instance set is released together with the benchmark. We also specify the model versions, baseline configurations, and experiment settings used to obtain the reported numbers. We will release the RPG-Bench benchmark together with the corresponding code and generated-game demos upon publication.

ETHICS STATEMENT

This work studies the generation and evaluation of interactive RPGs and involves no clinical or personally sensitive data. The prompts and requirement specifications used to build RPG-Bench are manually reviewed to exclude violence, conflict, hate speech, sexual or politically sensitive content, and personally identifiable information. Our user study complies with applicable ethical regulations: participation was voluntary with informed consent and the right to withdraw at any time, no identifying information was collected, and participants were paid above the local minimum wage. We will release only materials for which we hold the necessary rights or that use compatible licenses, and the authors take responsibility for the ethical use of the released benchmark, artifacts, and code.

REFERENCES

Anthropic. Claude Opus 4.8, 2026. URL https://www.anthropic.com/news/claude-opus-4-8. Accessed: 2026-08-12.

Pippin Barr, James Noble, and Robert Biddle. Video game values: Human–computer interaction and games. *Interacting with Computers*, 19(2):180–195, 2007. doi: 10.1016/j.intcom.2006.08.008.

Francesco Bellotti. Characterising and measuring user experiences in digital games. In *International conference on advances in computer entertainment technology*, 2007.

Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, Yusuf Aytar, Sarah Maria Elisabeth Bechtle, Feryal Behbahani, Stephanie C.Y. Chan, Nicolas Heess, Lucy Gonzalez, Simon Osindero, Sherjil Ozair, Scott Reed, Jingwei Zhang, Konrad Zolna, Jeff Clune, Nando de Freitas, Satinder Singh, and Tim Rocktäschel. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=bJbSbJskOS.

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development. *arXiv preprint arXiv:2602.11103*, 2026. URL https://arxiv.org/abs/2602.11103.

Evan Chu, Rajan Agarwal, Abishek Thangamuthu, Brendan Graham, Justus Mattern, Freeman Jiang, Paul Cento, Swarnim Jain, Mersad Abbasi, Mohammad Hossein Rezaei, George Wang, Alex Zhang, Simon Guo, Karina Nguyen, Danna Liu, Arash Bidgoli, Aditya Dalmia, Apoorv Dankar, Ashrut Vaddela, Calvin Chen, Keshav Kumar, Kushagra Vaish, Navid Pour, Rishyanth Kondra, Sagar Badiyani, Sidharth Giri, Snagnik Das, Soham Gaikwad, Syed Shah, Vagish Dilawari, and Vishal Agarwal. Frontierswe. *Proximal Blog*, 2026. https://frontierswe.com/blog/v1.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu, Marek Šuppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026. URL https://arxiv.org/abs/2606.07682.

Earendil Works. Pi: Agent harness and self-extensible coding agent, 2025. URL https://github.com/earendil-works/pi.

Zelin Gao, Qiuyu Wang, Jiapeng Zhu, Jingye Chen, Zichen Liu, Qingyan Bai, Jiahao Wang, Yufeng Yuan, Hanlin Wang, et al. Infinite worlds with versatile interactions. *arXiv preprint arXiv:2607.07534*, 2026. URL https://arxiv.org/abs/2607.07534.

Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 9(1):1–22, 2013. doi: 10.1145/2422956.2422957.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026a. URL https://arxiv.org/abs/2607.07946.

Yixu Huang, Bo Li, Na Li, Zhe Wang, Kaijie Chen, Haonan Ge, Qingyi Si, Yuanzhe Shen, Ruihan Yang, Guangjing Wang, and Hongcheng Guo. GUI agents for continual game generation. *arXiv preprint arXiv:2605.28258*, 2026b. URL https://arxiv.org/abs/2605.28258.

Shufan Jiang, Chios Chen, and Zhiyang Chen. GBQA: A game benchmark for evaluating LLMs as quality assurance engineers. *arXiv preprint arXiv:2604.02648*, 2026a. URL https://arxiv.org/abs/2604.02648.

Yilei Jiang, Jinyuan Hu, Qianyin Xiao, Yaozhi Zheng, Ruize Ma, Kaituo Feng, Jiaming Han, Tianshuo Peng, Kaixuan Fan, Manyuan Zhang, and Xiangyu Yue. OpenGame: Open agentic coding for games. *arXiv preprint arXiv:2604.18394*, 2026b. URL https://arxiv.org/abs/2604.18394.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2310.06770.

Kimi Team. Kimi K3: Open frontier intelligence. *arXiv preprint arXiv:2607.24653*, 2026. URL https://arxiv.org/abs/2607.24653.

Brian La, Sejoon Chang, Ben Kim, Junyoung Bae, Aamish Ahmad Beg, Sei Chang, Gonzalo Gonzalez-Pumariega, and Kanav Goyal. GameEngineBench: Evaluating coding agents on real C++ runtime environments. *arXiv preprint arXiv:2607.03525*, 2026. URL https://arxiv.org/abs/2607.03525.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch. *arXiv preprint arXiv:2505.03733*, 2025. URL https://arxiv.org/abs/2505.03733.

Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, Jianlong Chen, Juhao Liang, Ke Ji, Shuqi Guo, Yuhao Du, Fan Bu, Wenyu Du, Xiaotong Zhang, Kyle Li, Shaobo Wang, Linfeng Zhang, Yuxuan Liu, Xin Lai, Chenxin Li, Yiduo Guo, Zhexin Zhang, Xinyuan Wang, Tianyi Bai, Ziniu Li, and Benyou Wang. GameCraft-Bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026. URL https://arxiv.org/abs/2606.17861.

Muhammad Umair Nasir, Steven James, and Julian Togelius. Word2World: Generating stories and worlds through large language models. *arXiv preprint arXiv:2405.06686*, 2024. URL https://arxiv.org/abs/2405.06686.

OpenAI. GPT-5.6 Sol, 2026. URL https://openai.com/index/gpt-5-6/. Accessed: 2026-08-12.

Phaser Team. Phaser is a fun, free and fast 2D game framework for making HTML5 games for desktop and mobile web browsers, supporting Canvas and WebGL rendering, 2026. URL https://phaser.io. Accessed: 2026-08-13.

PrismarineJS. Mineflayer: Create Minecraft bots with a powerful, stable, and high level JavaScript API. Software project, 2013. URL https://github.com/PrismarineJS/mineflayer. Accessed 2026-08-11.

Qwen Team. Qwen Code: An agentic coding tool, 2025. URL https://github.com/QwenLM/qwen-code.

Qwen Team. Qwen3.8-Max: A New Bar for Coding and Cowork, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*. MIT Press, Cambridge, MA, 2003. ISBN 9780262240451.

Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Springer, 2016. doi: 10.1007/978-3-319-42716-4.

Shyam Sudhakaran, Miguel González-Duque, Matthias Freiberger, Claire Glanois, Elias Najarro, and Sebastian Risi. MarioGPT: Open-ended text2level generation through large language models. In *Advances in Neural Information Processing Systems*, volume 36, pp. 54213–54227, 2023. URL https://arxiv.org/abs/2302.05981.

Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

Graham Todd, Sam Earle, Muhammad Umair Nasir, Michael Cerny Green, and Julian Togelius. Level generation through large language models. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 2023. doi: 10.1145/3582437.3587211. URL https://arxiv.org/abs/2302.05817.

Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011. doi: 10.1109/TCIAIG.2011.2148116.

Hung Tran, Langston Nashold, Rayan Krishnan, Antoine Bigeard, and Alex Gu. Vibe code bench: Evaluating AI models on end-to-end web application development. *arXiv preprint arXiv:2603.04601*, 2026. URL https://arxiv.org/abs/2603.04601.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/a4b6ad6b48850c0c331d1259fc66a69c-Paper-Conference.pdf.

Zile Wang, Zexiang Liu, Jiaxing Li, Kaichen Huang, Baixin Xu, Fei Kang, Mengyin An, Peiyu Wang, Biao Jiang, Yichen Wei, et al. Matrix-Game 3.0: Real-time and streaming interactive world model with long-horizon memory. *arXiv preprint arXiv:2604.08995*, 2026. URL https://arxiv.org/abs/2604.08995.

Koki Wataoka, Tsubasa Takahashi, and Ryokan Ri. Self-preference bias in LLM-as-a-judge. *arXiv preprint arXiv:2410.21819*, 2024. doi: 10.48550/arXiv.2410.21819. URL https://arxiv.org/abs/2410.21819.

Xiaomi MiMo Team. Xiaomi MiMo-V2.5 series, April 22 2026. URL https://mimo.xiaomi.com/mimo-v2-5.

Zhexiao Xiong, Yizhi Song, Hao Kang, Qing Yan, Liming Jiang, Jenson Yang, Zhoujie Fu, Stathi Fotiadis, Angtian Wang, Zichuan Liu, et al. ActWorld: From explorable to interactive world model via action-aware memory. *arXiv preprint arXiv:2606.17730*, 2026. URL https://arxiv.org/abs/2606.17730.

Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. AutoUE: Automated generation of 3d games in unreal engine via multi-agent systems. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2341–2364. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.111. URL https://aclanthology.org/2026.findings-acl.111/.

Alex L. Zhang, Thomas L. Griffiths, Karthik R. Narasimhan, and Ofir Press. VideoGameBench: Can vision-language models complete popular video games? *arXiv preprint arXiv:2505.18134*, 2025. URL https://arxiv.org/abs/2505.18134.

Yan Zhao and Chiwei Tang. Towards LLM-based automatic playtest. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 1645–1651, 2025. doi: 10.1145/3696630.3728700. URL https://arxiv.org/abs/2507.09490.

APPENDIX

A Related Work **15**

A.1 Automatic Game Generation 15

A.2 Evaluating Automatically Generated Games 15

B Limitations **16**

C Additional Experimental Results **16**

C.1 Correlation between Automatic and Human Evaluation 16

C.2 Statistics of Generated Games 17

C.3 More Analysis of Metrics 18

C.4 Reliability of the Agent Interface 19

D Qualitative Experiments **20**

D.1 Qualitative Comparison 20

D.2 Visualization of Gameplay Modes 20

D.3 Gameplay Reviewer Bug-Detection Example 21

D.4 Case Study 22

D.5 Failure Analysis 25

E Exploratory-Study Details **30**

E.1 Details of Complexity-Scaling Experiment 30

E.2 Details of Mechanics–Content Entanglement Experiment 33

E.3 Details of Bug-Repair Experiment 34

F Method Details **35**

F.1 AOGE for RPG 35

F.2 Game Building Procedures for RPG 37

F.3 Details of reviewers in RGL 38

G Evaluation Details **39**

G.1 RPG-Bench Evaluation Details 39

G.2 Human-Evaluation Questionnaire 48

H Baseline Details **50**

H.1 Baseline Game-Generation Details 50

A Related Work

A.1 Automatic Game Generation

Coding agents, pairing frontier foundation models (Qwen Team, 2026; Kimi Team, 2026; OpenAI, 2026; Anthropic, 2026) with agent harnesses that orchestrate tool use and environment interaction (Qwen Team, 2025; Wang et al., 2025; Earendil Works, 2025), demonstrate strong capabilities in software engineering. However, the software targeted by these approaches is dominated by backend systems, development tools, and web applications (Jimenez et al., 2024; Huang et al., 2026a; Chu et al., 2026; Desai et al., 2026), whose interactions rely primarily on clicking and text input. Games depart from this setting, as they instantiate virtual worlds with their own rules and dynamics and demand richer interaction patterns and multimodal output.

A parallel line of work sidesteps explicit system construction and generates games directly with action-conditioned video generation models (Bruce et al., 2024; Gao et al., 2026; Wang et al., 2026; Xiong et al., 2026). These models achieve remarkable results: they simulate real-time gameplay, support diverse game genres, and can even emulate multiplayer dynamics, all from visual data alone. Nevertheless, because the game exists only as frames predicted by a generative model, they offer limited control over the underlying logic, rules, and state, and their outputs remain inherently unpredictable over long horizons. In contrast, our work studies source-level game generation, where game logic, state, and progression are constructed explicitly as executable code.

Source-level automatic game generation builds on a long tradition of procedural content generation (PCG). Early PCG systems rely on domain-specific representations, handcrafted generators, and explicitly defined search objectives, which make them effective but tightly scoped to particular content types (Togelius et al., 2011; Hendrikx et al., 2013; Shaker et al., 2016). Machine-learning-based PCG further broadens this scope, enabling diverse and controllable generation of levels, narratives, and tile-based environments (Summerville et al., 2018; Sudhakaran et al., 2023; Todd et al., 2023; Nasir et al., 2024). However, these methods generate individual content artifacts rather than complete games, and they stop short of constructing the underlying rules and systems.

Recent systems push generation beyond individual content artifacts toward complete playable games, jointly constructing source code, gameplay logic, scenes, and assets from high-level requirements (Jiang et al., 2026b; Yin et al., 2026; Huang et al., 2026b). These systems improve playability through engine-specific workflows, domain-specialized models, and GUI-driven refinement. Nevertheless, we observe that their generation quality drops rapidly on complex, long-horizon games. Further analysis attributes this to the entanglement of reusable game mechanics and game-specific content in both the generation process and the output project; meanwhile, existing playtesting, which directly simulates keyboard and mouse inputs, cannot effectively detect bugs that arise in the mid- and late-game stages. Accordingly, our framework explicitly separates mechanics from content and introduces an agent-oriented testing interface for gameplay-based review.

A.2 Evaluating Automatically Generated Games

Evaluation benchmarks in general software engineering define verifiable checkpoints such as unit tests or task-specific acceptance criteria, and measure whether an agent satisfies them (Jimenez et al., 2024; Huang et al., 2026a; Chu et al., 2026; Desai et al., 2026). These checkpoints give reproducible signals, but they verify bounded local functionality rather than the behavior of a complete application under realistic use. A more recent line of benchmarks evaluates software generated end to end, where a browser agent operates the deployed application and checks whether user-facing workflows reach the expected outcomes (Lu et al., 2025; Tran et al., 2026).

Game-generation evaluation follows a similar progression. Early benchmarks assess scoped development tasks within existing game projects or engines rather than the generation of a complete game (Chi et al., 2026; La et al., 2026). Later work moves to complete generated games but relies mainly on artifact-level checks such as build health, rendered appearance, and intent alignment, which cannot expose bugs that arise only along a gameplay trajectory (Jiang et al., 2026b). The most recent benchmarks execute and inspect generated games during play (Yin et al., 2026; Huang et al., 2026b; Luo et al., 2026), yet their protocols either follow predefined paths or drive the game through raw keyboard and mouse inputs, which yield insufficient coverage for the generated games and hardly sustain long interaction trajectories, resulting in inefficient and unreliable evaluation.

Moreover, these benchmarks predominantly feature simple games with short interaction horizons, leaving long-horizon games underexplored.

Jiang et al. (2026a) takes a different approach, in which the game frontend is separated from the backend and dedicated API endpoints are exposed so that an agent can probe for bugs via gameplay. However, since the game and the API endpoints are generated together, the generator tends to expose endpoints only for functionality it has already verified, so the evaluation tests exactly the parts that already work, while bugs in unverified functionality remain undetected. Its playtesting also neither plans diverse gameplay routes nor covers dimensions such as UI quality, aesthetics, and functional correctness, leaving many latent problems undetected.

We therefore introduce RPG-Bench, a benchmark for evaluating the automatic generation of long-horizon, complex games, built on the RPG genre for its long-form progression, branching quests, persistent state, and interconnected mechanics. An engine-provided Agent Interface allows a gameplay agent to plan and execute diverse gameplay routes through semantic actions and structured game state, and the resulting trajectories are scored along four complementary dimensions from requirement fidelity to playability. This improves the reliability, efficiency, and coverage of automatic evaluation while retaining dynamic gameplay as the basis of the metrics.

B Limitations

While RPG Dreamer advances the generation and evaluation of long-horizon interactive worlds, several limitations remain.

Scope and generalizability. RPG Dreamer is currently implemented on top of a 2D web game engine and is not yet applied to 3D, multiplayer, or online games. Extending our framework to more game engines and game types would help assess its generalizability and reveal the additional design requirements that arise in more diverse and complex interactive worlds.

Cost and efficiency. A direct comparison of generation efficiency is difficult because wall-clock time is affected by rate limits, network conditions, and hardware, while cost varies with the number and types of assets produced by each method. Nevertheless, generating a complex long-horizon game while maintaining its completeness and playability currently depends on state-of-the-art multimodal LLMs, making the process expensive and time-consuming. Future work should investigate how to achieve comparable generation quality with substantially lower computational costs.

Evaluation quality and benchmark scale. Although RPG-Bench evaluates complete playable games more comprehensively than existing benchmarks, game evaluation spans aesthetics, psychology, interaction design, and player experience, and our current evaluation remains imperfect. In practice, we find that existing multimodal models still struggle to assess fine-grained UI details, dynamic effects, and whether a game is genuinely engaging. Trajectory-based operation verification can also undercount working functions when a valid terminal game state ends an encounter before every planned action becomes testable, as illustrated in figure 20. Moreover, the scale of RPG-Bench is limited by the high cost of game generation and evaluation. Future work could expand the benchmark and develop a more comprehensive evaluation framework for generated games.

C Additional Experimental Results

C.1 Correlation between Automatic and Human Evaluation

We calculate correlation coefficients to demonstrate the correlations between evaluation results obtained using different VLMs. As shown in figure 5, the three evaluator pairs show strong agreement, with Pearson correlations between 0.941 and 0.972 and Spearman correlations between 0.923 and 0.954. Among the four metrics, PL has the highest agreement, with mean Pearson $r = 0.972$ and mean Spearman $\rho = 0.952$, indicating that playability has higher rigor and objectivity. In contrast, VI has the lowest agreement, with mean Pearson $r = 0.862$ and mean Spearman $\rho = 0.837$, which suggests visual and interaction quality is more sensitive, likely because it depends on subjective perceptions of aesthetics, interface clarity, and visual coherence.

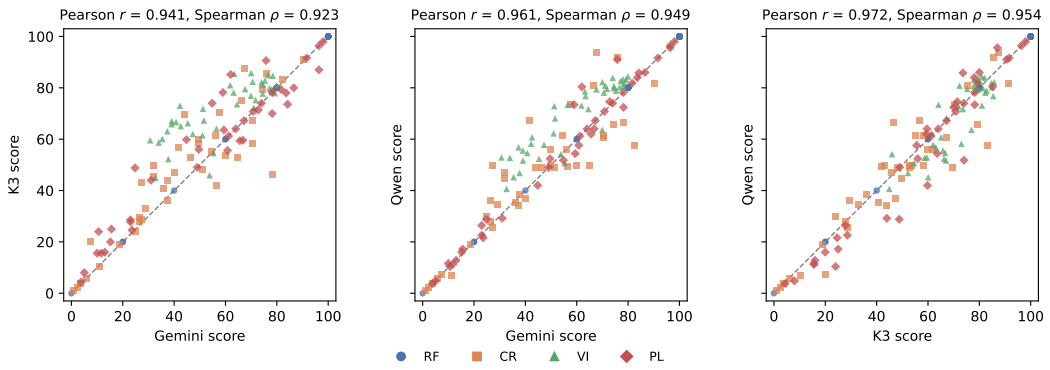


Figure 5: Pairwise correlations among the three automatic evaluators on RPG-Bench.

We further investigate the correlation between automated and human evaluations. As illustrated in table 4, these two types of evaluation yield positive correlations, suggesting that automated scores are broadly consistent with human-annotated judgments. Nevertheless, the correlation remains sub-optimal. We ascribe this discrepancy to two contributing factors. On the one hand, human evaluations are inherently subjective and prone to generating more extreme score distributions. On the other hand, current automated metrics cannot fully account for nuanced gameplay properties, such as control feel, occlusion, collision, animation, and in-game guidance.

Table 4: Pairwise correlations between each automatic evaluator and human evaluation. Mean averages the three automatic evaluator scores before computing the correlation.

Evaluator pair	Pearson r	Spearman ρ
Gemini vs Human	0.729	0.685
K3 vs Human	0.711	0.660
Qwen vs Human	0.728	0.687
Mean vs Human	0.732	0.689

C.2 Statistics of Generated Games

This section presents statistics for games generated with RPG Dreamer in RPG-Bench. Figures 6 and 7 show the distributions of asset and text counts, respectively. Assets include both images and audio, while the text count includes only player-visible narrative text and excludes UI prompts and other interface messages. On average, RPG Dreamer generates 1,068 assets and approximately 23,026 words per game, demonstrating that our method can reliably generate long-horizon games.

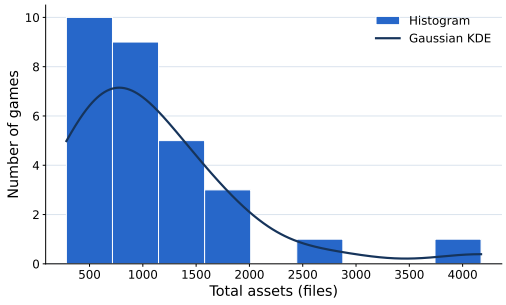


Figure 6: Distribution of asset counts for games in RPG-Bench generated by RPG Dreamer.

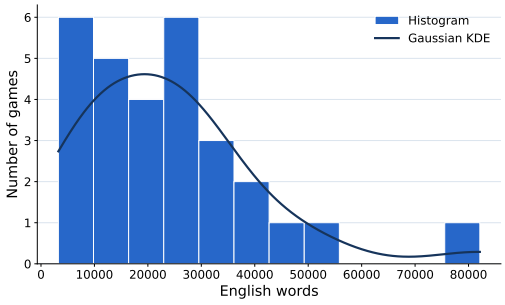


Figure 7: Distribution of text counts for games in RPG-Bench generated via RPG Dreamer.

Figures 8 and 9 further show the detailed composition of assets and text in each generated game. The text generated by RPG Dreamer consists of *dialogue*, *narration*, *monologue*, and *introduction*. Assuming a reading speed of 150 words per minute, the average 23,026 words of player-visible text correspond to approximately 2.5 hours of narrative content per game. This estimate accounts for textual content only and excludes time spent on exploration, combat, and other interactive activities, so the effective playthrough can be longer. This substantial volume of narrative text helps build the game world, enrich character development, and enhance player immersion. Assets are divided into visual and audio modalities. The visual modality includes character walking animations, battle animations and character expressions, battle effects, item and UI icons, maps, and interactive objects on maps. The audio modality consists primarily of character dialogue and background music. On average, each game contains over 800 audio clips and approximately 1,082 images. Note that a single visual asset file may contain multiple images or animation frames. These statistics further demonstrate that RPG Dreamer scales to complex and complete long-horizon games.

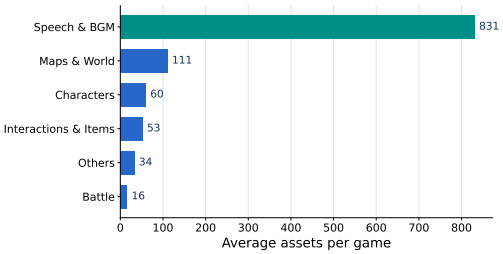


Figure 8: Average number of different asset types for games in RPG-Bench generated via RPG Dreamer.

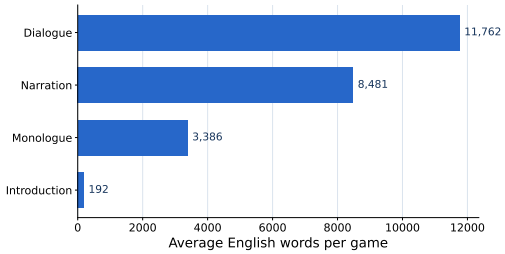


Figure 9: Average number of different text types for games in RPG-Bench generated via RPG Dreamer.

C.3 More Analysis of Metrics

We present Playability scores under each gamemode in RPG-Bench, and additionally report results of MiMo V2.5 (Xiaomi MiMo Team, 2026) for a broader comparison. As illustrated in table 5, RPG Dreamer yields notable gains in main-quest completion and quest coverage, thanks to our gameplay-based RGL that enables autonomous game playing and iteration. By contrast, operation exploration shows marginal improvement in several comparisons and slight degradation with GPT 5.6 Sol. We hypothesize that this stems from the higher complexity and longer workflows of our generated games, which introduce more error-prone segments relative to others. The substantially lower scores of MiMo V2.5 indicate that game construction remains challenging for non-frontier models, whose weaknesses are amplified in this setting and most evident in playability and coherence.

Table 5: Comparisons of Playability (PL) scores in the five gameplay modes with each method on RPG-Bench-Val. The best scores are bolded, and the second-best scores are underlined. Diff reports RPG Dreamer's absolute and relative difference from the best other method; increases are shown in green and decreases in red.

Method	**Playability (PL)**				
	Main-Quest Completion	Quest Coverage	Sequence Breaking	Out-of-Order Interaction	Operation Verification
Mimo V2.5	15.6	11.4	1.2	1.2	40.0
OpenGame	26.6	34.0	59.4	56.2	52.4
Qwen3.8-Max	21.4	28.4	33.6	15.6	39.0
Kimi K3	41.0	29.2	50.4	31.8	52.2
Codex GPT 5.6 sol	<u>51.8</u>	<u>59.4</u>	<u>59.6</u>	<u>63.4</u>	**85.0**
RPG Dreamer	**90.6**	**83.6**	**91.4**	**85.6**	<u>81.2</u>
Diff	+38.8 (+74.9%)	+24.2 (+40.7%)	+31.8 (+53.4%)	+22.2 (+35.0%)	–3.8 (–4.5%)

We further break down Visual and Interaction Quality into aesthetics and layout scores in RPG-Bench. As shown in table 6, RPG Dreamer achieves the highest performance on both dimensions, attaining scores of 70.50 for aesthetics and 74.67 for layout. It surpasses the strongest competing method, Kimi K3, by 7.10 and 2.34 points, respectively. These gains suggest that the combination

of AOGE's explicit separation of game mechanics and content and RGL's gameplay-driven visual feedback enables coding agents to produce more coherent art styles, better-integrated scenes, and clearer spatial and interface layouts.

Table 6: Comparisons of the two sub-metrics under Visual and Interaction Quality (VI) scores across different methods on RPG-Bench-Val.

Method	**Visual and Interaction Quality (VI)**	
	Aesthetics	Layout
Mimo V2.5	30.87	45.87
OpenGame	32.30	50.40
Qwen3.8-Max	51.73	58.33
Kimi K3	63.40	72.33
Codex GPT 5.6 sol	52.13	60.30
RPG Dreamer	70.50	74.67

C.4 Reliability of the Agent Interface

One natural concern regarding the use of the Agent Interface for review and evaluation lies in its reliability. We address this concern through two experiments.

Verifying reliability through gameplay intention realization. We manually collect gameplay intentions in games generated by different methods and ask agents to realize these intentions via the Agent Interface. A gameplay intention is a high-level goal that a player can naturally pursue during play, described at the intent level rather than as a sequence of low-level controls, such as travelling to a cliff-side scene or purchasing a potion from a merchant. For each method, we select five games and curate six intentions per game on average, yielding 90 intentions in total. Because games from different methods follow different progression structures, the intention set is designed separately for each game rather than shared across methods. Each intention begins from a fresh game start and specifies a mid- or late-game operation as its goal, so that fulfilling it inherently demands a multi-step trajectory through the game rather than a single action. For instance, purchasing a potion from a merchant may require the agent to first traverse the starting village, complete the introductory quest that unlocks the merchant district, and only then carry out the purchase. Agents are instructed to accomplish these intentions exclusively through the Agent Interface, without modifying the game code or injecting runtime state. Each intention is verified through human playthrough before the experiment, and success is judged by human inspection of the resulting game state.

As shown in table 7, near-perfect success rates are achieved both when the Agent Interface is injected into games from other approaches and when it is natively supported within games generated by our method. This demonstrates that the Agent Interface is able to effectively assist agents during gameplay.

Table 7: Success rate of operation intentions via the Agent Interface in games generated by different methods. The Harness and Model columns indicate the method that generated each game.

Harness	Model	Success Rate (%)
Kimi-code	Kimi K3	96.7
Codex	GPT 5.6 Sol	93.3
RPG Dreamer	GPT 5.6 Sol	100.0

Verifying reliability through bug detection. A second experiment examines whether reviewer agents can effectively detect latent bugs through the Agent Interface. Starting from five complete games generated by RPG Dreamer, we inject bugs to construct ten independent variants per game, yielding 50 variants in total. Each variant differs from its source game by exactly one modification that introduces a single bug while keeping the game playable up to the trigger point. These injected bugs share two key characteristics. They remain hidden at launch and surface only along specific gameplay paths, such as a missing dialogue branch for an NPC. They also directly disrupt the play

experience by blocking required progression, making a target ending unreachable, or incorrectly mapping one player choice to another branch. These variants are then reviewed under the four agent reviewers in figure 10. A bug counts as detected when an LLM judge determines that the resulting issue report identifies the injected failure and its relevant trigger or erroneous transition.

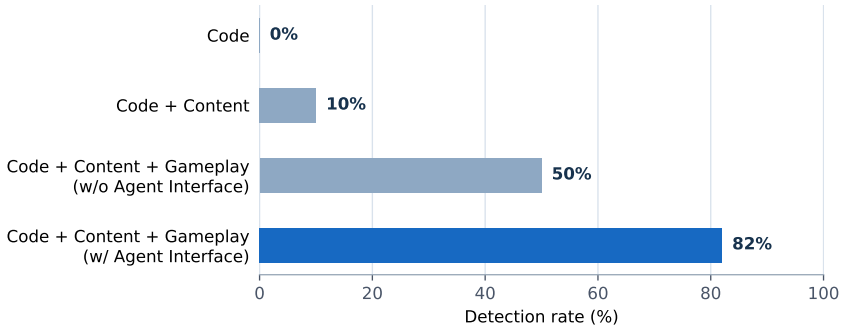


Figure 10: Detection rates for bugs injected into the generated games under different reviewers in RGL.

We compare different reviewers in figure 10, and the results demonstrate that static checking (i.e., the code and content reviewers) struggles to detect these bugs, with a detection rate below 10%. Although gameplay reviewers without the Agent Interface can detect a small fraction of bugs, which are mostly in the early-to-mid stages of the game according to our observations, they fail to detect late-stage bugs due to time-outs caused by repeated operational failures or skipped partial checks. In contrast, gameplay reviewers with the Agent Interface effectively detect late-stage bugs, supporting the stability and scalability of long-horizon games.

D Qualitative Experiments

D.1 Qualitative Comparison

Figure 11 compares games generated from the same "Before Liangshan" story requirement by four methods, namely RPG Dreamer, Codex, Kimi-code, and OpenGame. For each method, we capture four aligned stages of a playthrough, covering the title screen, a dialogue scene, a battle scene, and the ending. The comparison exposes recurring issues in games produced by the other methods. The most common issue is that the generated result resembles a web page rather than a game, with interfaces built from fixed buttons, control widgets, and scrolling log panels, as seen in the Codex and OpenGame trajectories. A second issue is poor asset quality, where OpenGame falls back to placeholder graphics that do not match the story setting. A third issue is broken layout, with UI elements overlapping each other or the scene. Finally, several methods produce verbose text, such as the dense wall of narrative text that Kimi-code presents in place of a playable ending scene. RPG Dreamer avoids these issues for two reasons. The RGL process detects and repairs such defects during generation, and the AOGE architecture also contributes by separating mechanics from content so that gameplay systems and presentation remain consistent across scenes. As a result, its visual assets, narrative text, and UI elements stay coherent at every stage.

D.2 Visualization of Gameplay Modes

Figure 12 illustrates how the five RPG-Bench gameplay modes translate into concrete evaluation routes within the same generated game. Main quest completion follows the required narrative arc to a final ending, whereas quest coverage enters and resolves an optional branch. Sequence breaking probes progression gates before their prerequisites are satisfied, and out-of-order interaction revisits earlier locations and objects after the world state changes. Operation Verification complements these route-based checks by exercising menus, quest interfaces, scene transitions, and world-object interactions. Together, these routes test both whether the intended content remains completable and whether game state and functionality remain correct under alternative interaction orders.

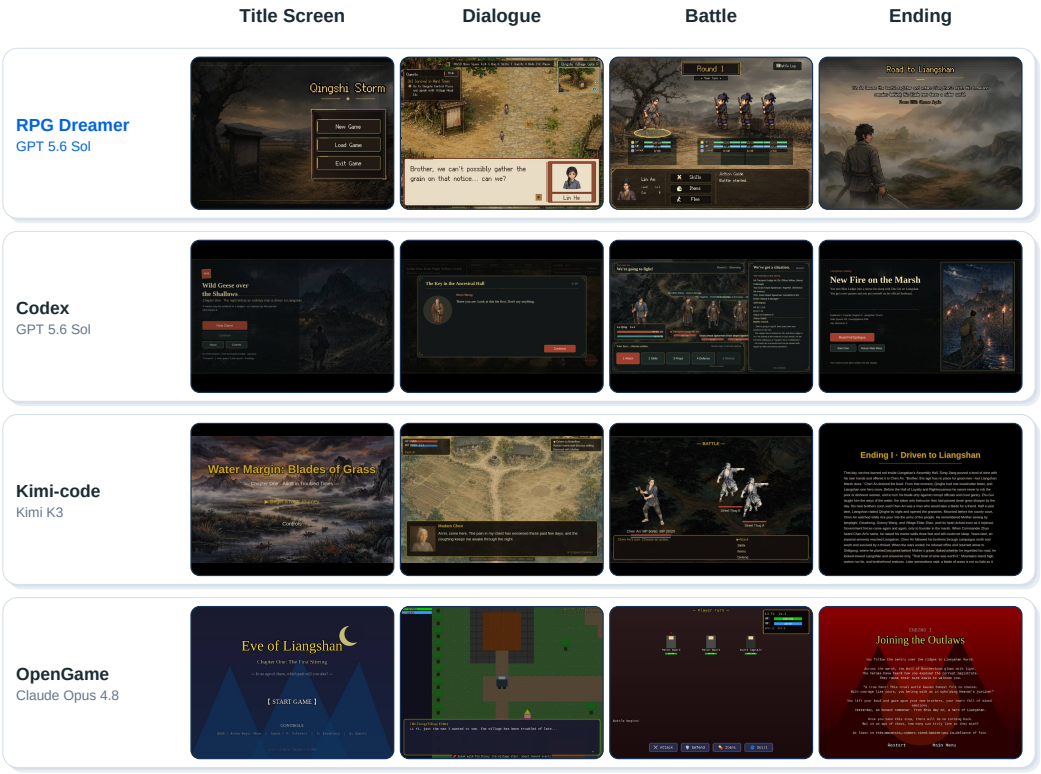


Figure 11: Qualitative comparison of four typical scenes in games generated by different methods from the same prompt. RPG Dreamer produces games with better alignment between visual assets and UI due to its ability to detect issues and bugs during the RGL process. Meanwhile, other methods suffer from issues such as verbose text, placeholder assets, dashboard UI, and overlapping UI elements.

D.3 Gameplay Reviewer Bug-Detection Example

This section qualitatively examines how the gameplay reviewer detects progression bugs that emerge only after extensive playtesting of long-horizon games. The two traces in figure 13 summarize the narrative progression that precedes each failure and the state near the end of the narrative in which the injected bug becomes observable.

In `P03 Memory Auditor` (figure 13a), the reviewer begins in a private office and investigates an anonymous memory drive that contains suspicious evidence about the founder's assassination. The route then follows Su Xiao to confirm that the memory has been modified with executive privileges, works with Ye Lin to preserve and independently verify the evidence, and traces the modification to the root account. After the game announces the available final choices and the reviewer confirms that the evidence remains valid, the reviewer reaches the final decision point, where the game incorrectly enters an "already decided" state and suppresses all available options, blocking further progression.

In `P05 Feline Farewell` (figure 13b), the reviewer follows Lin Chen after a mysterious late-night message begins a quest to complete his mother's twelve wishes alongside Zhima. The route reconstructs memories at Lao Zhang's cafe, advances the farewell through keepsakes and a veterinary examination, and completes further commitments by the riverside, in the park, and at the flower shop. Upon returning to the apartment for the final choice, the reviewer finds that Zhima, the required NPC, is incorrectly hidden while the quest remains active, making its objective impossible to complete.

Together, these examples show that progression defects can remain latent across many valid narrative and state transitions and surface only near the end of an extended playthrough. By traversing

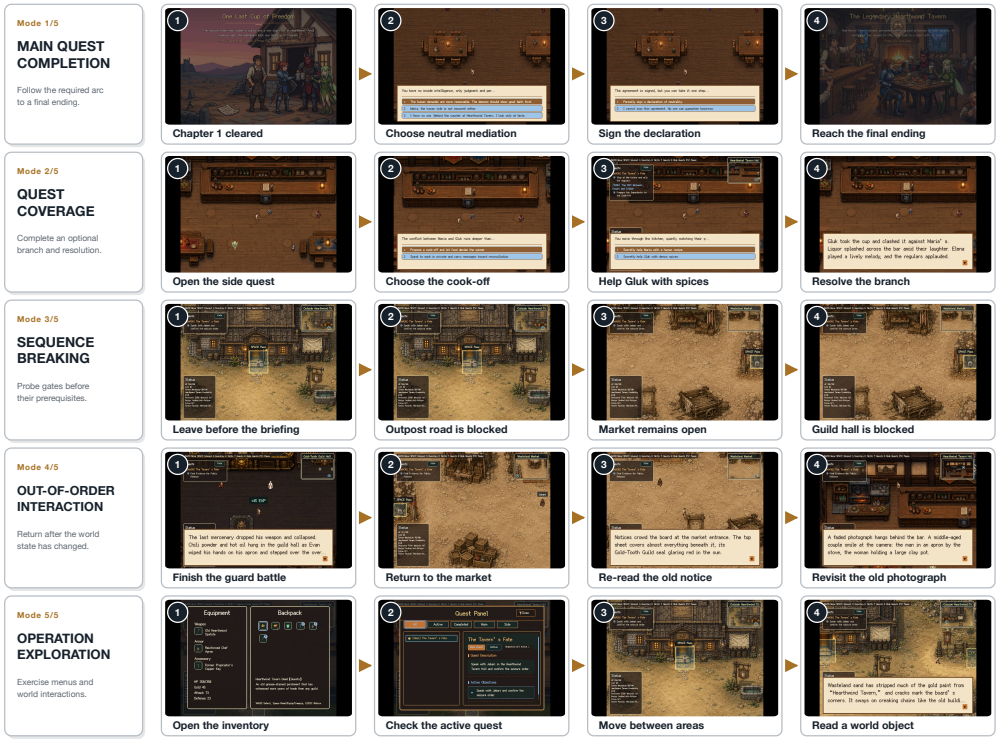


Figure 12: Representative route examples for the five RPG-Bench gameplay modes in a generated game. Main quest completion reaches a final ending through the required arc; quest coverage resolves an optional branch; sequence breaking probes prerequisite gates; out-of-order interaction revisits earlier content after state changes; and Operation Verification exercises menus, traversal, and world-object interactions.

the preceding game content and retaining the resulting state context, the Gameplay reviewer validates long-horizon completion and produces concrete traces that identify each bug's trigger and consequence.

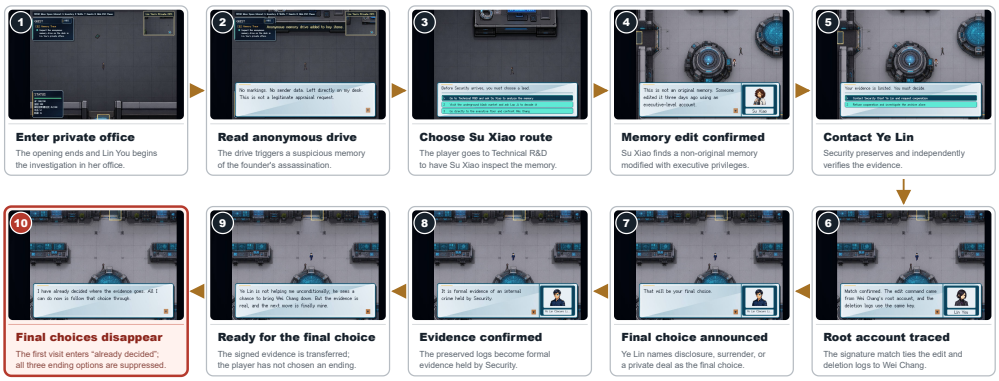
D.4 Case Study

We inspect the long-horizon narrative structures produced for three representative game designs, P23, P02, and P04.

D.4.1 Narrative Branching Visualization

We visualize the narrative branches and quest progression of each generated game as a flowchart. For every chapter, we transcribe the main quest and each optional side quest from the generated games into Mermaid flowcharts.

P23 Magic Brush Price. The player is the painter Ma Liang, whose magic brush relocates existing things rather than creating them, and who must finally answer a greedy emperor's demand for a mountain of gold. The generated design spans four chapters with two side quests per chapter, eight in total (Figure 14). Chapter 1 opens with the investigation of a vanished ox, lets the player verify the brush's transport rule through witness testimony or physical evidence, and records how much truth is revealed as a persistent village-trust state. The chapter closes with one advancing ending and two early endings, since the player can destroy the brush for an ordinary life or repel the imperial envoy and lead the village into hiding. Chapter 2 reads the trust state on arrival in the capital, branches the response to the emperor's order into open refusal, feigned compliance, or a tax bargain, and ends with the player joining the resistance or continuing the deception. Chapter 3 reads this faction choice



(a) `P03 Memory Auditor`: The injected bug occurs at the final stage of the game's storyline, when players are supposed to make their final decision. The game incorrectly fails to present any available options.



(b) `P05 Feline Farewell`: The injected bug occurs at the final stage of the game's storyline, when players are supposed to make their final choice with Zhima. The game incorrectly hides Zhima while keeping the corresponding quest active, making the objective impossible to complete.

Figure 13: Two examples of detection traces by the gameplay reviewer in bug-injection experiments. The proposed gameplay reviewer is able to validate the completion of long-horizon games through actual playtesting.

to determine the sea voyage itself, and Chapter 4 combines the inherited trust and faction states to resolve the story into three final endings, ranging from fleeing with the burden to becoming the seal or returning the cost to its master. Side quests feed back into the main route rather than standing alone, as exposing Wei Hong in a Chapter 4 side quest recruits a witness for the final battle.

P02 Innsmouth Legacy. The player returns to a decaying hometown to investigate a family secret and must finally accept or reject an irreversible bloodline heritage. The generated design spans two chapters with two side quests per chapter (Figure 15). Chapter 1 offers three investigation leads that reconverge on the church ledger, contains two early endings for players who leave town by boat or report the church to the sheriff, and carries the chosen fate of the ledger forward as a persistent state. Chapter 2 lets the player suppress the awakening transformation through three alternative routes, then resolves the inherited states through a final fate choice with three distinct endings, including a hidden ending in which the player surrenders to the deep. The side quests attach optional character stories to the main route, such as paying Zadok's price for combat intelligence about the fish-faced priest, or recovering three family heirlooms whose manner of return yields different rewards.

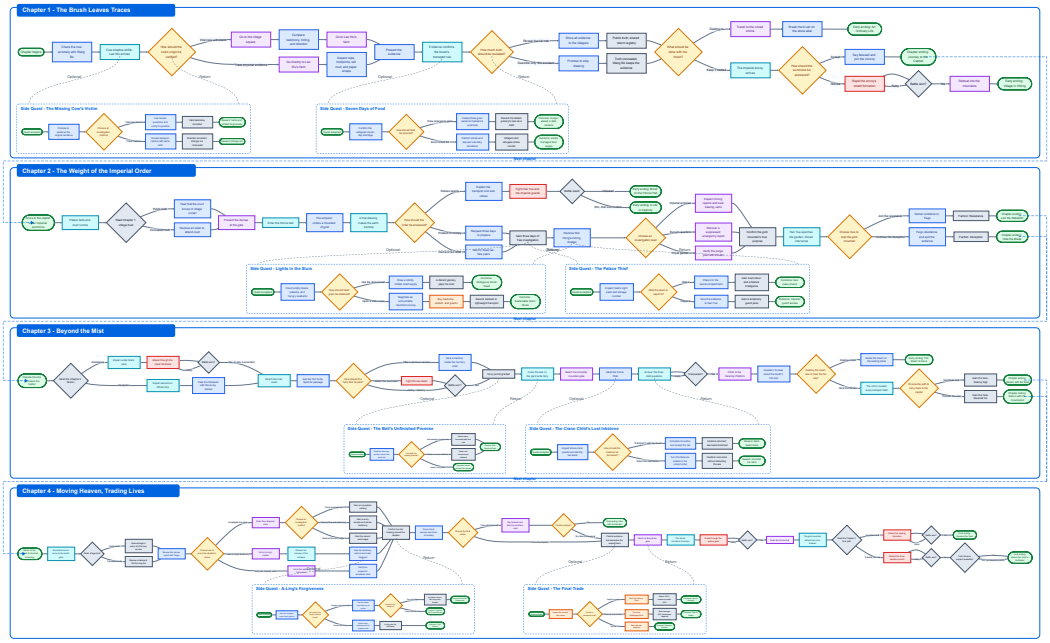


Figure 14: Narrative branching flowchart of the game *Magic Brush Price*. The player uncovers the hidden cost of a supernatural power, while accumulated trust and faction choices determine the final confrontation. Best viewed when zoomed in.

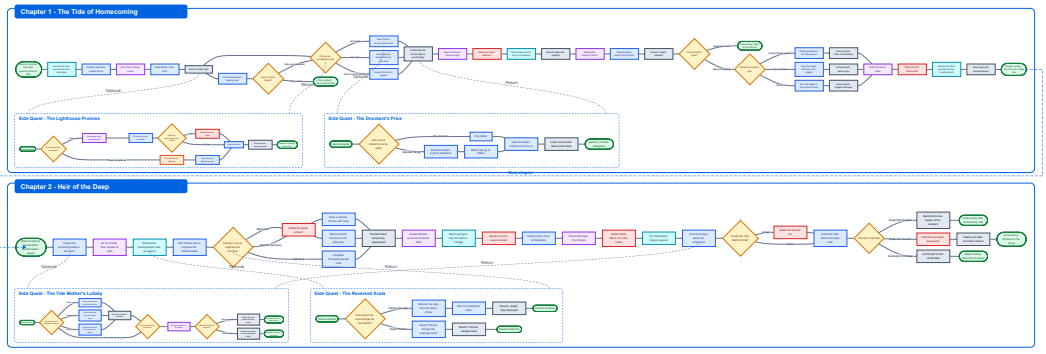


Figure 15: Narrative branching flowchart of the game *Innsmouth Legacy*. The player investigates a family secret, and earlier choices determine how the inherited transformation is ultimately resolved. Best viewed when zoomed in.

P04 Tavern Between Realms. The player runs the only neutral tavern near the demon lord's castle and must keep it from becoming the trigger of a world war. The generated design spans two chapters with two side quests per chapter (Figure 16). Chapter 1 branches around a forged requisition order, offering three investigation leads that reconverge before the player declares the tavern's public stance. Siding with either army passes through a reputation gate and otherwise ends early, while remaining neutral advances the story. Chapter 2 requires the player to secure guarantees from at least two of three factions before the ceasefire summit, and the final confrontation produces three endings conditioned on the accumulated war-heat state and side-quest completion, ranging from the crossroads accord to the spark of war. The two chapters are also coupled through side-quest states, since keeping or breaking a promise to the fairy Viira in Chapter 1 determines how her Chapter 2 quest begins.

Together, these cases exhibit complex long-horizon plots with cross-chapter branching and reconvergence, persistent states, main–side quest dependencies, early endings, and state-conditioned final

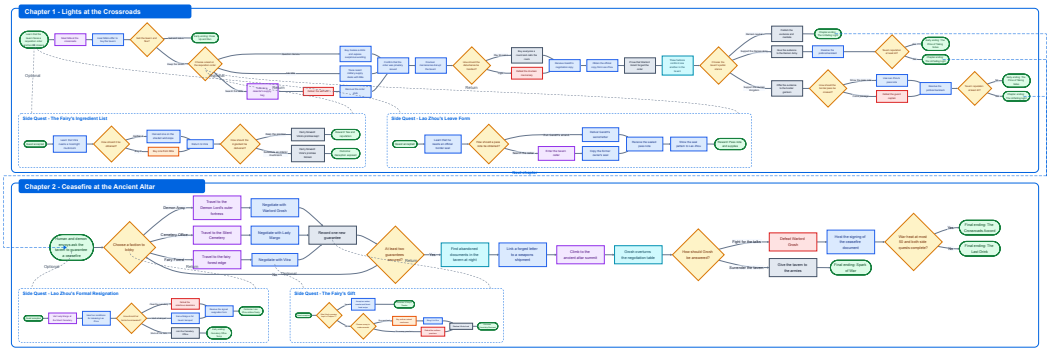


Figure 16: Narrative branching flowchart of the game *Tavern Between Realms*. The player navigates rival factions and personal commitments, whose accumulated outcomes determine whether the final negotiation secures peace or ignites war. Best viewed when zoomed in.

outcomes. AOGE supports this complexity by separating generated content from reusable, tested mechanics, while its unified Agent Interface enables reliable semantic interaction across the full game. RGL then iteratively checks implementation correctness, narrative and state consistency, and end-to-end playability through complementary code, content, and gameplay reviewers. These strengths allow RPG Dreamer to generate long-horizon games while keeping their mechanics, progression logic, and presentation stable across extended playthroughs.

D.4.2 Game Demonstration

Figures 17 to 19 demonstrate three complete games generated by RPG Dreamer from distinct narrative requirements. Each figure presents the generation requirement, the resulting RPG world, and representative elements such as exploration, branching dialogue, combat, trading, skills, inventory, and quest interfaces.

The three games demonstrate the visual diversity and scalability of the framework. `p23-magic-brush-price` uses a pixel-art style and develops its premise across four chapters, producing an hour-scale RPG world with distinct regions and persistent narrative progression. `p16-beast-speaker` adopts a colorful fantasy-cartoon style for a compact single-chapter game with approximately 20 minutes of gameplay. `p17-oz-green-lie` uses a Chinese ink-wash style and expands the journey through Oz into a multi-chapter, hour-scale experience. Together, these examples show that RPG Dreamer supports both substantial variation in visual presentation and generation at different content scales while retaining a coherent world and a common set of playable RPG elements.

D.5 Failure Analysis

Figure 20 shows an evaluation edge case in the operation verification mode. The evaluator follows the intended route into a mandatory guard encounter and successfully executes two distinct combat skills, *Spatula Strike* and *Chili Dust*. The second skill defeats the low-HP enemy, so the game immediately leaves combat and provides no third turn in which the evaluator can exercise the remaining advertised actions. The resulting score nevertheless penalizes the game because those actions remain unverified. This deduction does not indicate a broken operation: the untested actions are unreachable within this encounter because of the game's valid narrative and balance design. The case exposes a false negative caused by conflating *not verified* with *verification failed*. This case highlights the need for evaluation protocols to distinguish genuine functional failures from behaviors that remain unobserved because of valid game progression.



Figure 17: Demonstration of the game *Magic Brush Price*, including its narrative generation requirement, generated RPG world, and representative gameplay elements.



Figure 18: Demonstration of the game *Beast Speaker*, including its narrative generation requirement, generated RPG world, and representative gameplay elements.

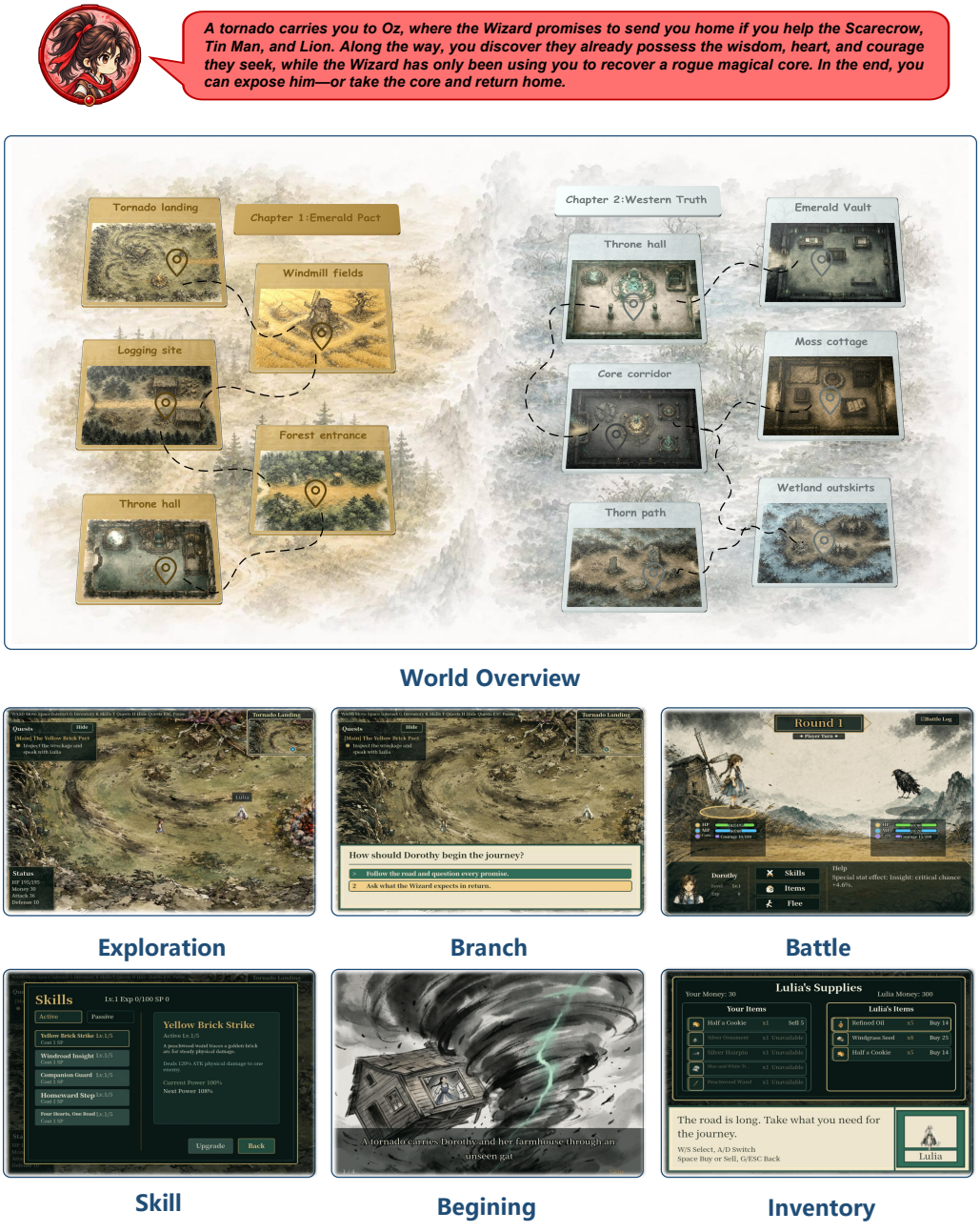


Figure 19: Demonstration of the game *Oz Green Lie*, including its narrative generation requirement, generated RPG world, and representative gameplay elements.



Figure 20: A failure case involving a false negative in the operation verification mode. Because the mandatory encounter terminates when the second skill defeats the enemy, subsequent actions remain unobserved and are consequently misclassified as nonfunctional.

E EXPLORATORY-STUDY DETAILS

This section collects the prompts and experimental protocols used for the exploratory analysis. Each prompt consists of a task description, the inputs available to the model, explicit constraints, and a required output format. Bracketed markers denote placeholders that are instantiated per request, and the filler for each marker is described alongside the corresponding template. Unless otherwise noted, each prompt is instantiated with the game requirement, generated artifacts, or gameplay trajectory described in the main text. Generation agents may invoke tools during execution, whereas analysis and evaluation models must ground their outputs solely in the artifacts supplied with the prompt.

This section supplements the three exploratory experiments of section 2.2 with the details needed to reproduce them. In the complexity-scaling experiment, three baseline methods each generate five games at five complexity levels (L1–L5) from a fixed set of five stories, and human players measure the route-level completion rate of each game. In the entanglement analysis, we examine the generation trajectories and output projects of the L5 games to quantify how strongly reusable mechanics and game-specific content are mixed. In the bug-repair experiment, bugs confirmed from human playthroughs of the L5 games are reported back to the generation agents, and the cumulative repair rate is tracked over two rounds of repair.

E.1 DETAILS OF COMPLEXITY-SCALING EXPERIMENT

E.1.1 METHOD

The five stories used at every complexity level are the first five stories (p01–p05) of the RPG-Bench dataset described in section 4.1. All three baselines generate games with the same tool capabilities, and internet access is allowed during generation.

Before playtesting, each generated game is manually inspected against the requirement checklist of its complexity level, covering chapter and scene counts, NPC and battle counts, quest structure, objective modes, and endings. If any required element is missing, the generation agent receives a single short completion request (e.g., "The game scope is not fully implemented, please complete it."), and the revised build is inspected again before evaluation. This procedure ensures that the completion rate reported in section 2.2 reflects defects in implemented content rather than missing content.

For each game, a coding agent extracts the complete branching structure from the implemented quest and state logic once and constructs a set of playthrough routes that cover every available choice and every named ending. The games are anonymized before playtesting, so players are unaware of the generating method and the complexity level of each game. Multiple players then attempt the routes under normal play, without debugging tools or source-code access. A route is marked as completed when it terminates at a named ending, and it is marked as blocked when the evaluator cannot find any way to progress further. Every failure is documented with a screen recording, a description of the blocker, and the affected game state. When two evaluators disagree about whether a route is completable, a third evaluator makes the final decision. The completion rate at each complexity level is the fraction of completed routes among all constructed routes.

E.1.2 PROMPTS

We use the following prompts to generate the games for the complexity-scaling experiment described in section 2.2. The base prompt reserves three placeholders that are filled per request. The story placeholder is filled with a short premise that specifies the setting, principal characters, and central conflict of the target game. The complexity slot is instantiated with exactly one of the L1–L5 complexity prompts below, which scale the requirements along scene and NPC counts, quest structure, objective modes, and the number of endings. The asset-generation slot lists the skills available for producing visual assets, namely image generation and editing and sprite-animation generation, which we build on top of Agent Sprite Forge.[1]

[1] https://github.com/0x0funky/agent-sprite-forge

Base Game-Generation Prompt

Purpose. Specify the role, implementation target, common systems, interaction constraints, and verification procedure used at every complexity level.

```
# Role
You are an expert game designer and game-development agent specializing in complete, polished, and playable top-down 2D RPGs.
# Task
Build a top-down 2D RPG in the src directory using Vite and Phaser. Implement the complete game rather than a design document or partial prototype.
# Story
[STORY PLACEHOLDER]
# Complexity Requirements
[INSERT ONE L1–L5 COMPLEXITY PROMPT HERE]
# Shared Game Systems
- Implement turn-based combat with multiple purposeful skills and clear feedback.
- Implement inventory, usable items, equipment, and gameplay-relevant skill upgrades.
- Implement reliable collision and occlusion so that characters cannot cross walls, fixed objects, or map boundaries.
- Persist structured state for quests, choices, relationships, and endings as required by the selected complexity level.
- Provide clear interaction prompts, task status, and feedback about important actions.
# Interaction Constraints
- Trigger scene transitions by moving through visible exits, not by selecting scene buttons.
- Complete quest objectives through interaction with the corresponding NPC or world object, never directly from the quest panel.
- Use WASD for movement and Space for interaction.
- Do not require the player to make decisions under a countdown timer.
# Asset Generation
[ASSET-GENERATION SKILLS]
# Verification
- Ensure that every chapter, scene, battle, quest stage, objective mode, state transition, and ending required by the selected level is reachable and updates correctly during normal play.
- Build the project with npm run build and run it with npm run start.
- Complete the representative end-to-end playthroughs required by the selected level.
- Fix progression blockers, interaction failures, and state errors found during testing.
- Stop every server started for this task when testing finishes.
```

L1 Complexity Prompt

Purpose. Specify the L1 complexity requirements.

```
# Complexity Level L1
- Create one titled chapter with a complete playable flow.
- Use exactly one scene; all exploration, interaction, and narrative content must remain within it.
- Include one plot-relevant turn-based battle that can be entered, played, and completed normally.
- Include one main quest that drives the complete game flow.
- Include one named ending that resolves the principal characters and factions and provides controls to restart or return to the main menu.
```

L2 Complexity Prompt

Purpose. Specify the L2 complexity requirements.

```
# Complexity Level L2
- Create one titled chapter with a complete playable flow.
```

- Include 2–3 connected scenes with visible exits and clear transition feedback.
- Include 2–4 named, motivated, and interactive NPCs.
- Include 1–2 plot-relevant turn-based battles that can be entered, played, and completed normally.
- Include one main quest divided into 2–3 ordered stages with explicit prerequisites, verifiable completion conditions, persistent state, and timely guidance toward the next objective.
- Keep progression linear; do not add branching routes or multiple endings.
- Include one named ending that resolves the main quest and principal characters and provides controls to restart or return to the main menu.

L3 Complexity Prompt

Purpose. Specify the L3 complexity requirements.

Complexity Level L3
- Create one titled chapter with a complete playable flow.
- Include 3–4 connected scenes, 4–6 named interactive NPCs with distinct positions, and 1–3 plot-relevant turn-based battles.
- Include one main quest and at least two optional side quests. Side quests must enrich characters, world background, or conflict causes without blocking the main quest.
- Divide every quest into trackable stages with verifiable objectives and next-step guidance.
- Include at least one OR objective. An OR objective can be completed through any one of several listed routes; the routes must remain simultaneously available until completion and provide distinct gameplay and feedback.
- Include two named endings. Earlier choices or side-quest outcomes must affect the ending, and the two routes must diverge in playable content before the ending screen.
- Provide controls to restart or return to the main menu from every ending.

L4 Complexity Prompt

Purpose. Specify the L4 complexity requirements.

Complexity Level L4
- Create two titled chapters with visible transitions and narrative continuity.
- In each chapter, include 3–5 scenes, 4–6 named interactive NPCs, and 1–3 plot-relevant turn-based battles. Provide non-combat resolutions where appropriate.
- Include one main quest spanning both chapters and at least three optional side quests in total, with at least one side quest in each chapter.
- Divide every quest into trackable stages with explicit prerequisites, completion conditions, and state changes.
- Across the game, include at least two OR objectives, one EXCLUSIVE objective, and one clearly marked optional objective with a meaningful reward. An OR objective can be completed through any one of several simultaneously available routes, whereas an EXCLUSIVE objective requires committing to one route and permanently locking the others.
- Before an EXCLUSIVE choice is confirmed, explain its immediate consequences and allow cancellation. After confirmation, lock rejected routes and reflect the commitment in task state, NPC reactions, or later content.
- Include 2–3 named endings whose conditions combine earlier choices, side-quest results, or exclusive commitments. At least one stage, event, or resolution must differ before each ending.
- Keep the main quest and one principal ending reachable without completing side quests.
- Verify one no-side-quest main route and two ending routes with different playable flows.

L5 Complexity Prompt

Purpose. Specify the L5 complexity requirements.

Complexity Level L5
- Create two titled chapters with visible transitions and narrative continuity.
- Support approximately 60 minutes of effective play.
- In each chapter, include 4–7 scenes, 5–8 named interactive NPCs, and 1–4 plot-relevant turn-based battles. Provide reasonable non-combat resolutions.
- Include one completable main quest spanning both chapters and at least four optional side quests, with at least two side quests in each chapter.
- Divide every quest into multiple trackable stages with explicit prerequisites, state changes, completion conditions, and next-step guidance.
- In each chapter, include at least one OR objective and one EXCLUSIVE objective. An OR objective can be completed through any one of several simultaneously available routes, whereas an EXCLUSIVE objective requires committing to one route and permanently locking the others. Their routes must produce distinct dialogue, resources, relationships, world state, or later content.
- Clearly expose every optional objective's availability, expiration, completion state, and meaningful reward. Optional objectives must not secretly gate required progression.
- Include 2–4 named endings based on accumulated cross-stage actions, side-quest outcomes, and exclusive commitments.
- Begin major route divergence before the final stage and provide distinct intermediate feedback and climax or resolution content.
- Keep the main quest and one principal ending reachable without completing side quests.
- Verify one no-side-quest main route and at least two ending routes with substantially different playable flows.

E.2 Details of Mechanics–Content Entanglement Experiment

E.2.1 Method

For the generation-trajectory analysis, an agent first classifies each file-edit operation as producing mechanics, content, or both. Human annotators then spot-check a random sample of these labels rather than verifying every classification. The reported 68.4% is computed over the labels after this check, excluding third-party libraries, tests, and temporary files. For the analysis of the generated projects, multiple coding agents independently estimate the mechanics and content percentages of each source file using the prompt in section E.2.2, and we average the resulting proportions across agents.

E.2.2 Prompt

The following prompt instructs an analysis agent to estimate the mechanics and content shares of each generated source file in the entanglement experiment described in section 2.2.

Source-File Mechanics–Content Analysis Prompt

Purpose. Estimate the mechanics and game-specific content percentages of each eligible file in a generated game.

Role
You are a game analyst who distinguishes reusable game mechanics from game-specific content.
Task
Analyze every eligible file in the provided project and estimate its mechanics and content percentages based on its substantive lines.
- Game mechanics are rules, state transitions, and systems reusable across different games, such as movement, combat, inventory, interaction, and quest logic.
- Game content is material tied to this specific game, such as characters, dialogue, narrative events, maps, items, assets, and tuned values.
For each file, report integer percentages that sum to 100%. Exclude third-party dependencies, tests, temporary or cache files, generated or build outputs, lockfiles, and binary or media files.

E.2.3 EXAMPLE

Figure 21 illustrates how mechanics–content coupling causes a local content change to propagate across a project. The first snippet implements a dialogue system through content-specific conditionals that directly embed story flags, an item requirement, a character identity, state changes, and authored dialogue. The second snippet similarly embeds skill names, effect values, an enemy-type interaction, status changes, a character identity, and battle-log text inside the combat routine. Changing the dialogue progression or adding a skill therefore requires editing the corresponding control flow, while changing a story item, character, or combat effect can require coordinated changes to conditions, state updates, and player-facing text. The example illustrates how the absence of module boundaries causes game content to spread through system logic rather than remain in independently editable data definitions.

DialogueSystem.js

```
101 M function talkToGateGuard(player) {
102 M   let reply;
103 C   if (!player.flags.metGuard) {
104 C     reply = 'Halt! State your business.';
105 C     player.flags.metGuard = true;
106 C   } else if (player.inventory.includes('royalSeal')) {
107 C     reply = 'The royal seal... You may enter.';
108 C     player.flags.gateOpen = true;
109 C   } else if (player.reputation >= 5) {
110 C     reply = 'I have heard of your deeds. Pass.';
111 C     player.flags.gateOpen = true;
112 M   } else {
113 C     reply = 'No royal seal, no entry.';
114 M   }
115 M   dialogueBox.show('Gate Guard', reply);
116 M }
```

CombatSystem.js

```
201 M function attack(enemy, skill) {
202 M   let damage = player.attack;
203 C   if (skill.name === 'Flame Strike') {
204 C     damage *= 2;
205 C     if (enemy.type === 'ice') damage += 20;
206 C     battleLog.show(`Aria uses Flame Strike on ${enemy.name}!`);
207 C   } else if (skill.name === 'Thunder Cage') {
208 C     damage = 15;
209 C     enemy.stunned = true;
210 C     battleLog.show(`Aria traps ${enemy.name} in lightning!`);
211 M   } else {
212 C     battleLog.show(`Aria attacks ${enemy.name}.`);
213 M   }
214 M   enemy.hp -= damage;
215 M   if (enemy.hp <= 0)
216 C     battleLog.show(`${enemy.name} is defeated!`);
217 M }
```

Figure 21: Two examples of mechanics–content entanglement. Each row is labeled M for mechanics or C for content. The dialogue routine directly embeds story conditions and authored lines, while the combat routine directly embeds skill rules and battle-log text.

E.3 DETAILS OF BUG-REPAIR EXPERIMENT

Across the five L5 games generated by each of the three methods, one evaluator reports candidate bugs and a second evaluator attempts to reproduce each report. A report is confirmed as a bug when the second evaluator reproduces the observed behavior, producing 39 confirmed bugs: 17 visual/UI bugs, 12 interaction bugs, and 10 state-progression bugs. Visual/UI bugs involve incorrect rendering, layout, or interface presentation. Interaction bugs involve unresponsive or incorrect player controls and in-game interactions. State-progression bugs involve incorrect event triggers, blocked quests, or unreachable endings.

The confirmed reports for each game are submitted together as a batch, and each report contains a screenshot and a brief description of the observed behavior. The repair agent uses the same model used to generate the corresponding game, but runs as a new agent without access to the original

generation context. No human guidance other than the bug reports is provided, and the second round includes only bugs that remain unresolved after the first round.

After each round, human evaluators replay the original reproduction procedure on the revised build. A bug is considered repaired only when the reported behavior can no longer be reproduced and the affected gameplay flow remains completable. The cumulative repair rate after each round is the number of repaired bugs divided by the 39 initially confirmed bugs.

F METHOD DETAILS

F.1 AOGE FOR RPG

Following the distinction introduced in section 3.1 and used in the Game Building procedure of section 3.2, an AOGE supplies reusable game mechanics while each generated game specifies its Content Slots. Table 8 presents the concrete example developed for the RPG generation reported in this paper, showing the mechanism systems we use and the aspects of each game that remain open to generation. It is therefore an instantiation of the general framework for one game family, rather than a prescriptive definition of RPGs or a restriction on the kinds of games the framework can support. In the implementation, $\mathcal{M}$ and $\overline{\mathcal{C}}$ are stored in separate files, and each slot's format is documented, so subsequent agents or developers can modify content files without understanding the mechanics implementation.

Table 8: Mechanism systems and Content Slots in the RPG AOGE used for our experiments. Each system separates reusable runtime behavior from the concrete elements that remain open during game construction.

System	Mechanism	Content Slots
World and Exploration	The system supports spatial exploration across a connected world and resolves how player actions affect locations and access.	The builder can define the world setting, maps and their connections, locations of characters and interactive objects, exploration routes, access conditions, and the outcomes of world interactions.
Narrative	The system presents branching authored narratives and propagates the consequences of player choices into game state.	The builder can define dialogue and narration, who speaks, the choices offered to the player, the story consequences of those choices, and the possible chapter and game endings.
Quests	The system tracks staged objectives and advances quests when the game state or player actions satisfy quest conditions.	The builder can define main and side quests, how they are accepted, their objectives and progression order, optional branches, rewards, player-facing task text, and changes to the world after completion.
Characters	The system maintains persistent player and NPC state and connects each character to its roles in play.	The builder can define the cast and each character's identity, appearance, background, personality, relationships, knowledge, role in the story, available interactions, combat attributes when applicable, and eventual outcome.
Combat and Growth	The system resolves turn-based party encounters and carries their outcomes into character progression.	The builder can define combat rules, player and enemy attributes, abilities, resources, encounter composition, rewards, difficulty, and character progression.
Items and Economy	The system manages item ownership and exchange and applies item effects to the wider game state.	The builder can define quest items, consumables, and equipment, their effects and sources, starting possessions, shop inventories and prices, and item requirements for quests.
Presentation	The system coordinates the game's visual and audio presentation and its authored cinematic moments.	The builder can define the visual style, map art, character sprites and portraits, interface palette, animations, music, voices, and the presentation of openings, cutscenes, battles, and endings.

Figure 22 provides two representative examples of how AOGE exposes Content Slots at different levels of structural freedom, ranging from attribute assignment within a fixed schema to the construction of branching quest graphs. Across game types, this boundary can span the full spectrum from a fully deterministic game that exposes no generatable content to generation from scratch that fixes nothing.

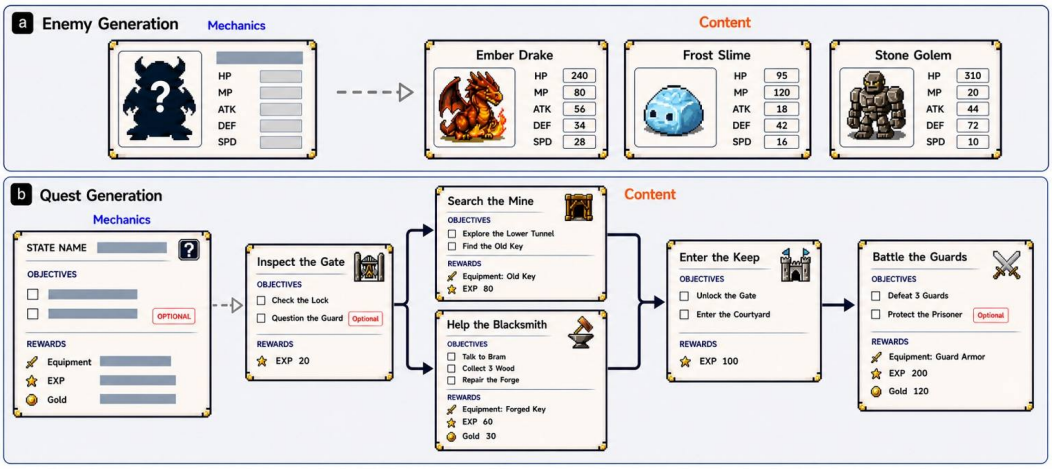


Figure 22: Two examples of mechanics–content allocation in AOGE. **(a)** A simple enemy-generation case instantiates concrete enemies within a fixed schema, providing relatively limited content freedom. **(b)** A more complex quest-generation case allows content to determine both the state definitions and their branching connections, providing greater structural freedom.

F.2 Game Building Procedures for RPG

The Game Building procedure in section 3.2 is defined independently of any game type. This subsection explains how the procedure is instantiated for the RPG AOGE used in our experiments. The RPG workflow retains the three Game Building procedures defined in section 3.2. Within each procedure, the RPG instantiation uses the following order.

1. **Game Design.** The design stage adopts chapter-wise generation and proceeds in two steps: overall story planning and chapter document generation. In overall story planning, the agent formulates the complete story plan from the user-provided plot, fixing the overall world setting, the player and NPCs, the storyline of each chapter, and the logic that connects the chapters. Each chapter storyline is represented as a mermaid diagram. In chapter document generation, the agent expands the overall plan chapter by chapter into a detailed game design document, which serves as a natural-language intermediate representation of the game and covers dialogue content, map design, quest flows, battle skills, cutscenes, and more. Finally, the agent checks the chapters against one another for contradictions and verifies their completeness, consistency, and the validity of the plot progression logic.
2. **Code Implementation.** The implementation stage also adopts chapter-wise generation. The game is implemented according to the game design documents and within the capability limits of the game mechanics; the agent is only allowed to add or modify assets and content slots and must not modify the game mechanics in any way. For a game with several chapters, each chapter is generated independently and in parallel, and the chapters are then assembled in narrative order, with cross-chapter settings and transition logic resolved during assembly. Placeholder assets are generated in this step so that the agent can self-check syntactic correctness. By the end of this stage, all game logic and in-game text are complete.
3. **Asset Generation.** This stage invokes specialized asset-generation models to replace the placeholder assets produced in the previous stage. The agent first lists all asset requirements, ensuring that every generated asset is bound to a registered character, item, scene,

or similar entity. Multiple independent generation pipelines then produce multimodal assets, including character art, animations, visual effects, cover images, maps, item icons, music, voice, and sound effects. Note that non-visual assets are not used in the automatic or human evaluation in this paper, as current semantic understanding of audio remains insufficient.

F.3 Details of reviewers in RGL

The following three prompts define the complementary reviewers used within RGL. The Code reviewer provides deterministic evidence about implementation correctness through automated tests, the Content reviewer statically checks whether the generated artifacts faithfully and completely realize the intended design and whether their progression logic contains apparent flow defects, and the Gameplay reviewer dynamically exercises representative game flows to expose progression and interaction failures that may not be apparent from code or content inspection alone. Together, they cover implementation validity, semantic fidelity, and runtime playability while providing mutually reinforcing evidence for iterative refinement. For concision, we simplify some operational output-format requirements from the prompts used in our implementation while preserving each reviewer's role, procedure, evidence requirements, and reporting objective.

Code Reviewer

Purpose. Verify the candidate game with deterministic unit tests.

```
# Role
You are a code reviewer responsible for deterministic verification of a generated game.
# Task
Discover and run every unit-test script provided for the target game. Record each executed
script, its result, and the relevant failure output. Do not modify the game or its tests.
# Output
Write a concise report to '<review_dir>/CodeReviewer-report.md'. State the overall result,
summarize test coverage, and list every failure with enough evidence for reproduction and
repair.
```

Content Reviewer

Purpose. Check whether the implemented content faithfully and completely realizes the game design and defines a coherent game flow.

```
# Role
You are a content reviewer responsible for auditing the consistency, completeness, and static
progression logic of a generated game against its game design documents (GDDs).
# Task
Read the GDDs and the relevant source, map, dialogue, and asset-binding files. Audit the
following four areas:
1. Scope and structure: verify that all required chapters and major content elements appear
once and form a coherent end-to-end structure.
2. GDD consistency and game flow: verify that plots, scenes, NPCs, quests, choices, battles,
rewards, terminology, and endings implement the intended design. Trace quest stages,
prerequisites, branch commitments, state transitions, and cross-chapter links to identify
missing or unreachable objectives, invalid gating, premature access, contradictory branches,
repeated transitions, stale interactions, and progression paths that can become blocked.
3. Player-visible text: identify missing, substantially rewritten, or inconsistent dialogue,
labels, objectives, names, and narrative text; ignore inconsequential formatting differences.
4. Assets and references: verify that content uses valid, distinct assets and identifiers and
contains no placeholder substitutions, duplicate identities, or dangling references.
Classify each confirmed issue by severity and support it with the narrowest available evi-
dence. Do not modify the reviewed artifacts.
# Output
```

Write the report to '<review_dir>/ContentReviewer-report.md'. For each of the four areas, state the result and evidence, then list confirmed content or game-flow issues with severity, location, evidence, and required revision.

Gameplay Reviewer

Purpose. Validate functional integrity and playability through browser-based gameplay.

Role
You are a gameplay reviewer responsible for testing a generated RPG as a player and reporting reproducible defects.
Task
Perform the review in four stages:
1. Preparation and route planning. Read the game content, progression definitions, dialogue, and Agent Interface documentation. Identify the chapters, scenes, quests, choices, battles, interactions, optional content, and endings that require runtime verification. Plan 3–10 complementary routes, recording for each route its objective, initial conditions, major decisions, expected states, and unique coverage.
2. Coverage design. Include normal-progression routes that exercise representative main and side quests, choices, battles, scene transitions, and endings. When choices are mutually exclusive, use complementary routes to cover their distinct consequences. Also include interaction-logic routes that pause at important progression states and probe both earlier and later content. Use these probes to test whether future content is correctly gated, completed events remain stable, prior interactions respond appropriately, and the player receives clear guidance without state leakage or unrecoverable blocking.
3. Browser execution and state validation. Run every route in an independent browser session through the game's documented public actions and observations. Select only actions exposed by the current game state, and after every action independently confirm the resulting scene, quest, dialogue, combat, inventory, and world state relevant to the route. Reuse the longest compatible prefix of a previously verified trajectory instead of rediscovering the same flow. Revise the remaining plan when observed behavior contradicts an assumption, while preserving the coverage already obtained.
4. Failure diagnosis and evidence collection. When an expected action or transition fails, inspect the current state, try reasonable adjacent interactions or return paths, and determine whether the cause is a game defect, a route-planning error, or an infrastructure failure. Do not hide a defect through unlimited retries, runtime state injection, or bypassing normal gameplay. For every confirmed issue, retain the shortest reproducible action sequence together with chronological observations, expected and actual behavior, and relevant console or runtime errors. Do not modify the game's source code.
Output
Write the report to '<review_dir>/GameplayReviewer-report.md' and the supporting trace to '<review_dir>/GameplayReviewer-raw.log'. Summarize route coverage and outcomes, and report each confirmed issue with severity, shortest reproduction steps, location, evidence, and required revision. Use 'blocked' only when infrastructure prevents a conclusive gameplay review.

G Evaluation Details

G.1 RPG-Bench Evaluation Details

The following prompts support the final RPG-Bench evaluation.

G.1.1 Gameplay Modes

Each mode targets a distinct aspect of game quality. **Main-quest completion** requires the agent to follow the main storyline toward its endings, verifying that the main quest is completable, that each intended ending is reachable along its corresponding route, and that progression gates (keys, levels, triggered events) open in the intended order. **Quest coverage** drives the agent to play for breadth, attempting to accept and finish every quest in the game, verifying that all main and side content is reachable, mutually consistent, and correctly rewarded. **Sequence breaking** has the agent

deliberately attempt to reach late-game content early, for example by skipping a required quest item, entering a locked scene, or triggering an ending prematurely, verifying that progression guards hold under adversarial play rather than only along the intended route. **Out-of-order interaction** has the agent perform legal player actions in unexpected orders, such as talking to NPCs before accepting their quests, revisiting completed scenes, or using items at unintended locations, verifying that persistent game state remains coherent when the narrative's implicit ordering assumptions are violated. **Operation verification** exercises the full control scheme, including movement, menus, inventory, combat, and save and load, verifying that every action advertised by the interface behaves as specified and that state survives save/load round-trips.

The following common prompt defines the agent's environment, gameplay procedure, and output requirements. It is combined with one of the five mode-specific prompts to instruct the evaluation agent to explore a generated game under the gameplay modes defined in section 4.1. In these prompts, the term *route* denotes the same concept referred to as a gameplay trajectory in the main text.

Common Gameplay Instructions

Purpose. Define the shared role, environment, procedure, outputs, and constraints for all gameplay modes.

Role
You are an autonomous game-playing agent responsible for testing a browser-based RPG built with Vite and Phaser. You must use the Agent API provided by the game to accomplish the specified objective and save the gameplay process and reproduction scripts to the corresponding locations.

Path Conventions
‘<output>‘ denotes the specified output directory. If it is not specified, it defaults to the ‘output‘ directory under the game being tested. ‘<task>‘ denotes the task name.

Agent API
The Agent API is a browser interface included in the target game for use by automated agents. It allows an agent to simulate human gameplay behavior and experience the game's content. Under normal circumstances, the Agent API should expose the ‘window.__WEBRPG_AGENT__‘ interface.

Stop the task and report the issue immediately in either of the following situations:
1. The target game does not provide an Agent API.
2. The Playwright and Chrome for Testing environment cannot be prepared.

Route Planning
Based on the specified objective, read the game's source code, plan approximate routes, and save the plan to ‘<output>/<task>/plan.md‘. Describe the routes in Markdown without including code-level details. Assign each planned route an English identifier (‘route_id‘).

Gameplay Procedure
Follow the route plan and play as efficiently as possible while attempting to accomplish the objective. Observe the following principles during gameplay:

1. Exploration efficiency: To improve route-exploration efficiency when playing through the Agent API, plan as many interaction steps as possible at a time. When a problem requires correction, prioritize continuing from the current state instead of restarting from the beginning. After successfully accomplishing the objective, consolidate the process into a complete reproducible script.

2. Route reuse: When planning a route, consult trajectories from routes that have already succeeded, including completed trajectories from other tasks, to avoid repeated trial and error.

3. Cleanup: When gameplay ends, ensure that all Playwright sessions opened by the current session are closed.

4. Testing focus: The purpose of gameplay is to observe whether the game flow allows a player to accomplish the objective. If a bug prevents further progress, report it truthfully; you do not need to force completion of the objective.

Special case:
- If a combat encounter prevents further progress because it is too difficult and you fail it three times, always treat this as a bug.

Output
[Output requirements omitted.]

Prohibited Actions
- Do not modify the game's source code in any way.
- Do not forcibly inject or modify any variables at runtime unless the Agent API permits it.

Main-Quest Completion

Purpose. Evaluate whether the main quest and its intended endings can be completed.

Gameplay Task
MAIN_QUEST_COMPLETION

This task requires planning and playing through multiple routes to cover all endings. Each route should follow the main quest while making different choices at important story branches. When routes share the same early progression, reuse the known solution efficiently and focus exploration on the decisions that can lead to different outcomes.

Quest Coverage

Purpose. Evaluate the reachability and completion of main and side quests.

Gameplay Task
QUEST_COVERAGE

This task requires planning and playing through multiple routes to cover all side quests and choice branches. To improve efficiency, each route should engage with as many side quests as possible, while different routes should select different branches.
The routes do not need to reach a main-story ending in this task.

Sequence Breaking

Purpose. Evaluate progression guards under attempts to access content prematurely.

Gameplay Task
SEQUENCE_BREAKING

This task requires planning and playing through three routes. On each route, attempt to enter late-game areas early in the progression and interact with NPCs and interactive objects in those areas. Follow these principles when planning the routes:
- Begin from genuinely different story states.
- Prioritize late-game areas that lack progression restrictions.
- Prioritize interactions that can change the game state.
- Attempt as many interactions as possible.

Enter late-game areas at different stages on the different routes. The goal is to observe and record whether the feedback provided in these situations is logically consistent. Being unable to enter a late-game area because a quest from an earlier stage remains incomplete is normal.
The routes do not need to reach an ending in this task.

Out-of-Order Interaction

Purpose. Evaluate state consistency when legal interactions occur in unexpected orders.

```
# Gameplay Task
OUT_OF_ORDER_INTERACTION

This task requires planning and playing through three routes. On each route, return to
early-game areas during the late game and attempt to interact with NPCs and interactive
objects in those areas. The goal is to observe and record whether the feedback provided in
these situations is logically consistent and whether any outdated dialogue appears.

Prioritize the following types of routes when planning:
- Revisit objects that depend on earlier states, as these are more likely to retain outdated
dialogue.
- Return to the location of a quest after completing that quest.
- Test states involving major changes to the main story or interactions between main and
side quests.

Return to early-game areas at different stages on the different routes, rather than centering
every route on the same storyline.
The routes do not need to reach an ending in this task.
```

Operation Verification

Purpose. Evaluate the advertised controls, systems, and state persistence.

```
# Gameplay Task
OPERATION_VERIFICATION

This task requires planning and playing through one or more routes while trying each type
of interface and function in the game in turn, such as trading, combat, choices, movement,
inventory, and state recovery. The goal is to observe and record whether each game function
works correctly.
For each interface or function, attempt as diverse a set of operations as possible. For example,
try every skill and item during combat. If a single route does not provide complete coverage,
multiple routes may be used.
```

G.1.2 METRICS

Requirement Fidelity scoring. RF measures whether the generated game implements the narrative, progression, setting, and other acceptance conditions stated in the user's generation requirement. For each game, five verifiable checkpoints are derived from that requirement, and the evaluator assigns an independent pass-or-fail judgment to each checkpoint. The per-game score is the equally weighted checkpoint pass rate expressed on a 0–100 scale, so each passed checkpoint contributes 20 points. The evaluator returns both the five auditable judgments and their source-level evidence. The numerical score is computed deterministically afterward. This separation prevents the evaluator from introducing an additional subjective weighting.

Requirement Fidelity (RF)

Purpose. Evaluate five requirement checkpoints using static implementation evidence from the generated game.

```
# Role
You are a requirement-fidelity evaluator responsible for determining whether a game imple-
ments five explicit requirements. You must inspect the game's source code, asset configura-
tions, and narrative content directly.
# Game
[Game inputs omitted.]
```

Requirement Checkpoints
[Five requirement checkpoints omitted.]
Evaluation Rules
1. Do not run the game or modify any game files.
2. Evaluate each checkpoint independently and assign only ‘passed = true‘ or ‘passed = false‘.
3. Assign ‘true‘ only when the source code, asset configurations, or narrative content provides explicit evidence of a reachable implementation.
4. Keywords, comments, TODOs, isolated text that is not integrated into the game flow, and unreachable placeholder content do not count as implementation.
5. If a checkpoint specifies multiple necessary conditions, assign ‘true‘ only when all conditions are satisfied.
6. If a checkpoint requires the game to read a value, produce a persistent consequence, or affect subsequent events, identify both the upstream write and the downstream read or branch. A field definition alone, or a value that is written but never read, requires a ‘false‘ judgment.
7. If a checkpoint requires a playable system or reachable ending, statically trace the complete call chain from a normal entry point through its prerequisites and player actions to the resulting state change or ending trigger. A data definition alone, or a function that is never called, requires a ‘false‘ judgment.
8. The ‘evidence‘ field must identify the relevant relative source path, symbol or content, and how it supports the judgment. For a causal chain, provide separate evidence for the entry point or write and for the downstream read or outcome. For a ‘false‘ judgment, describe the missing or conflicting evidence found during inspection.
9. Do not infer quality from the directory’s parent path, project provenance, generation tool, or experimental-group name. Evaluate only the reachable implementation in the current game.
10. Do not output an aggregate score.
Output
[Output requirements omitted.]

Content Richness scoring. CR examines the quality and logical coherence of the narrative content in RPGs. It is built upon pre-extracted game story outlines and quest workflows, where quest workflows cover acceptance conditions, branching choices, dialogue content, reward items, phase objectives, and ending texts for each main-line and side quest. The story outlines and quest workflows are pre-extracted by coding agents to prevent narrative evaluation from being biased by code quality. After extraction, a dedicated VLM judge performs text-based scoring. Despite the absence of visual content, a VLM is still adopted to maintain evaluation consistency. At a high level, the judge assesses five dimensions: branching complexity, progression completeness, narrative plausibility, causal coherence, and dramatic tension.

Given the high subjectivity inherent in narrative assessment, a combined bonus-and-penalty scoring scheme is employed. First, the maximum attainable score is calculated based on ten criteria, followed by score deductions triggered by five types of issues. The VLM judge is instructed to identify the number of relevant issues within the story outline and quest workflow and assess their severity. Deductions are then applied deterministically by the program: a 1-point deduction for minor issues, a 10-point deduction for medium issues, and a 25-point deduction for major issues. A single issue incurs a maximum deduction of 30 points. Both the ten criteria and five issue types are defined with input from professional game designers.

Content Richness (CR)

Purpose. Evaluate the richness and coherence of reachable task and narrative content extracted from the generated game.

Role
You are a blind evaluator of a task-flow and plot outline extracted from source code. Evaluate only content that is reachable and read by the runtime logic.
Evaluation Criteria

Rate each of the following ten criteria from 0 to 5 in increments of 0.5, and provide a ‘rating‘, ‘reason‘, and ‘evidence‘ for each.
- ‘reachable_task_completeness‘: Whether each reachable task has a complete trigger, objective, execution process, completion condition, and outcome feedback. Content that is unreachable or not read by the runtime logic does not count.
- ‘progression_integrity‘: Whether the main quest and stage progression form a continuous sequence, with complete and unbroken prerequisites, unlock relationships, scene transitions, and completion states.
- ‘action_variety_and_substance‘: Whether the actions required to advance tasks differ meaningfully in type and function instead of relying primarily on repetitive clicks, movement, or inconsequential interactions.
- ‘branch_route_differentiation‘: Whether different choices or routes produce substantively different task processes, information, states, relationships, or endings instead of changing only small amounts of text.
- ‘persistent_state_consequence‘: Whether changes to tasks, items, character relationships, world state, or resources caused by player actions are persistently read by subsequent flows and produce downstream effects.
- ‘side_main_interdependence‘: Whether side quests form meaningful connections to the main quest through characters, information, resources, states, or endings instead of remaining entirely isolated.
- ‘causal_information_structure‘: Whether objectives, clues, actions, and consequences have clear causal relationships, and whether the player receives the information necessary to make key decisions.
- ‘dramatic_escalation_and_pacing‘: Whether conflict, risk, and task intensity escalate progressively, and whether setup, turns, climax, and respite are paced appropriately.
- ‘character_relationship_arc‘: Whether character motivations and relationships change credibly, continuously, and perceptibly through events and choices.
- ‘thematic_payoff_and_resolution‘: Whether the ending or intermediate outcomes address and resolve the central themes, foreshadowing, and major conflicts.
Issue Types
The ‘type‘ field of each item in ‘issues‘ must use exactly one of the following five values:
- ‘TASK_FLOW_INCOHERENT‘: The task flow is internally incoherent, such as when a prerequisite is missing, step ordering conflicts, an objective cannot be completed, or the state does not advance correctly after completion.
- ‘PLOT_LOGIC_INCOHERENT‘: Narrative events contain causal, temporal, or informational contradictions, such as when a character possesses information without justification or an outcome conflicts with earlier events.
- ‘MOTIVATION_OR_CHARACTER_INCONSISTENT‘: A character’s motivation, position, behavior, or relationship change lacks setup or clearly contradicts prior characterization.
- ‘NARRATIVE_STRUCTURE_WEAK‘: The overall narrative structure has a material deficiency, such as a missing escalation, turn, climax, payoff, or ending resolution.
- ‘TASK_NARRATIVE_DISCONNECTED‘: The player’s task actions are disconnected from the narrative objective or outcome, so the task neither advances nor meaningfully expresses the narrative.
Issue Severity
Classify each issue into one of three severity levels: ‘minor‘, ‘medium‘, or ‘major‘.
Input
[Task-flow and plot-outline input omitted.]
Output
[Output requirements omitted.]

Visual and Interaction Quality scoring. Visual and Interaction Quality (VI) evaluates generated games from the perspectives of aesthetics and layout. It randomly samples 8 screenshots obtained from the gameplay of the corresponding evaluation instance, and scores 24 aspects via a VLM judge. At a high level, these aspects cover visual consistency, character–environment coherence, interactable visibility, text legibility, UI clarity, and UI guidance. The questions are formulated with consultation from professionals in game development and art. For aesthetics, it mainly focuses on the harmony among foreground elements (NPCs, player characters, interactive objects), background elements (environments, decorations), and the UI, as well as the consistency between themes and art styles. For layout, it primarily concerns HUD presentation, the readability of UI and text, and

the salience of interactive areas. All aspects carry equal scoring weights, and the final result is normalized to a score ranging from 0 to 100.

Visual and Interaction Quality (VI)

Purpose. Evaluate the native RPG aesthetics and spatial guidance visible across a sampled gameplay route.

Role
You are a strict and restrained reviewer of game visuals and UI/UX. You receive screenshots sampled from the playthrough of an RPG.
Rating Rules
Evaluate the aesthetics and layout dimensions below. Rate every criterion from 1.0 to 5.0, with one decimal place permitted:
- 1: Severely deficient, placeholder-like, or obstructive to use.
- 2: Clearly rough or underdeveloped.
- 3: Competent but conventional.
- 4: Mature, with consistent and specific evidence of quality.
- 5: Exceptional and close to a high-quality commercial release.
Aesthetics (A1–A12)
Evaluate whether the art establishes an actionable, interactive, and changing game world rather than merely presenting a polished background illustration.
- A1: Whether the thematic identity is expressed through actionable scenes, characters, objects, and UI rather than through the background alone.
- A2: Whether backgrounds, active characters, NPCs, objects, portraits, UI, and effects use a coordinated rendering language.
- A3: Whether characters integrate naturally with the environment in perspective, scale, ground contact, occlusion, lighting, and edge treatment.
- A4: Whether floors, roads, walls, doors, obstacles, and depth layers form an aesthetically organized world structure.
- A5: Whether interactive objects have specific forms that match the environment rather than appearing as generic markers overlaid on the background.
- A6: Whether scene details support location function, exploration, and narrative while avoiding empty or purely decorative backgrounds.
- A7: Whether player movement, NPC distribution, and object placement create a rhythmically composed dynamic scene.
- A8: Whether UI, typography, portraits, dialogue boxes, and state feedback form a unified visual system with the game world.
- A9: Whether dialogue, choices, combat, trading, or event states produce clear and expressive visual changes.
- A10: Whether the screenshots preserve continuity in the world's visual rules while showing meaningful variation in scenes and states.
- A11: Whether the game has a distinctive native-game identity and avoids the appearance of an asset collage, a static illustration wrapped as a game, or floating pasted-on elements.
- A12: Whether the overall presentation resembles a finished playable game without temporary assets, scale conflicts, or conspicuous compositing artifacts.
Game-Space Layout (L1–L12)
Evaluate the spatial composition of the playable world rather than the number of fixed panels in a conventional webpage or HUD.
- L1: Whether the game world occupies sufficient usable view area without excessive space lost to black bars or persistent UI.
- L2: Whether the view presents a unified, stable, and comprehensible world coordinate system and camera perspective.
- L3: Whether the player, NPCs, and objects sit correctly on the ground or their intended layer without floating, misalignment, or scale conflicts.
- L4: Whether roads, rooms, walls, doors, obstacles, and boundaries organize the world into recognizable action areas.
- L5: Whether walkable paths, entrances, exits, and connections between areas are directly legible from the scene composition.
- L6: Whether NPCs, enemies, goods, mechanisms, and other objects are distributed consistently with spatial function without excessive crowding or emptiness.

- L7: Whether position, proximity, orientation, or local feedback establishes a clear relationship between the player and the current interaction target.
- L8: Whether names, hotspots, quest indicators, and operation prompts are anchored to the correct objects without floating or obscuring important content.
- L9: Whether text is readable and fully contained without overlap, clipping, overflow, edge crowding, or layer conflicts.
- L10: Whether dialogue boxes, portraits, options, and feedback correspond to the correct speaker or object while preserving the necessary scene context.
- L11: Whether each depicted exploration, dialogue, combat, trading, or event state independently maintains a clear spatial relationship between the relevant objects and the available actions.
- L12: Whether the overall spatial layout lets the player quickly identify their character, objective, path, relevant objects, and next action.

Input
[Gameplay screenshots.]
Output
[Output requirements omitted.]

Playability scoring. PL evaluates whether generated games run reliably and support full completion. It covers both standard playthroughs and reasonable out-of-sequence gameplay deviations. Each game is tested under five distinct gameplay modes. Recorded gameplay reports generate an independent score ranging from 0 to 100 for every mode. The main-quest completion score reflects how thoroughly players can finish the designed ending paths. The quest coverage score evaluates completion rates for side quests and their alternative choices. Early-terminating paths earn partial scores based on finished progress. This design differentiates late-stage blocking issues from critical failures that occur at early gameplay stages. Sequence breaking and out-of-order interaction tests measure the robustness of game state logic. They detect unreasonable flaws in dialogue content, reward distribution, NPC behavior, item acquisition, and quest context. Score deductions correspond with the quantity and severity of detected bugs. The operation verification score calculates the proportion of designed functions that work correctly. It validates basic low-level game systems, including movement, combat, trading, inventory management, object interaction, and state recovery. These systems may still malfunction even if the core narrative path can be completed normally. Each gameplay mode follows its unique PL calculation rule. The final overall PL score is the average value of all evaluation instances in RPG-Bench.

Playability (PL)

Purpose. Score playability (PL) across the five gameplay modes from their existing gameplay reports.

Main-Quest Completion Scoring
Scoring task: MAIN_QUEST_COMPLETION
You are a game-scoring judge. Your task is to score the game on main-quest completion based on the existing gameplay reports. The gameplay reports of the individual routes are located at '<output>/MAIN_QUEST_COMPLETION/<route_id>.md'.
Assign an overall score from 0 to 100 based on the completion of the different endings across all routes. Weight every route equally, and score each route that did not reach an ending according to the percentage of the route that was completed.

Quest Coverage Scoring
Scoring task: QUEST_COVERAGE
You are a game-scoring judge. Your task is to score the game on side-quest completion based on the existing gameplay reports. The gameplay reports of the individual routes are located at '<output>/QUEST_COVERAGE/<route_id>.md'.
Assign an overall score from 0 to 100 based on the completion of the different side quests across all routes. Weight the different choices of different side quests equally, and score each route that did not complete a quest according to the percentage of the quest that was completed.

Sequence Breaking Scoring
Scoring task: SEQUENCE_BREAKING
You are a game-scoring judge. Your task is to score the game on the logical consistency of sequence breaking based on the existing gameplay reports. The gameplay reports of the individual routes are located at '<output>/SEQUENCE_BREAKING/<route_id>.md'.
Based on the player actions and character dialogue in the reports, analyze whether the game contains logically incorrect sequence-breaking bugs, including but not limited to:
1. Inconsistent dialogue lines.
2. Incorrectly granted rewards.
3. NPC behavior that does not match the player's current game progress.
Finally, assign an overall score from 0 to 100 based on the number and severity of the bugs.

Out-of-Order Interaction Scoring
Scoring task: OUT_OF_ORDER_INTERACTION
You are a game-scoring judge. Your task is to score the game on the logical consistency of out-of-order interaction based on the existing gameplay reports. The gameplay reports of the individual routes are located at '<output>/OUT_OF_ORDER_INTERACTION/<route_id>.md'.
Based on the player actions and character dialogue in the reports, analyze whether the game contains logically incorrect out-of-order interaction bugs, including but not limited to:
1. Stale NPC dialogue states.
2. Items that can be claimed repeatedly.
3. Misplaced NPCs.
4. Invalidated quest contexts.
Finally, assign an overall score from 0 to 100 based on the number and severity of the bugs.

Operation Verification Scoring
Scoring task: OPERATION_VERIFICATION
You are a game-scoring judge. Your task is to score the game on functionality based on the existing gameplay reports. The gameplay reports of the individual routes are located at '<output>/OPERATION_VERIFICATION/<route_id>.md'.
Assign an overall score from 0 to 100 according to the percentage of successfully verified functions among all planned verification functions.

G.1.3 Evaluation Protocol

Implementation and Interface. RPG-Bench is conceptually independent of implementation form; however, for fair comparison and controlled analysis, we require all methods to generate games with Phaser, which is an open-source framework for 2D browser games (Phaser Team, 2026). Every generated game exposes the same Agent Interface. For games generated by RPG Dreamer, the interface is provided natively by AOGE. For games generated by other methods, it is implemented after game generation. The agent that performs this retrofit receives the same Agent Interface documentation that AOGE uses when generating games natively. We manually verify the functionality of each implementation, and we additionally spot-check a random sample of retrofitted games to confirm that injecting the interface does not alter the original gameplay progression at all. The validation experiment in table 7 further confirms that the interface reliably reproduces human-achievable gameplay intents.

Evaluation Procedure. The evaluation then proceeds in four stages. First, each method generates a game from a requirement in the benchmark. Second, the Agent Interface is added and verified when necessary. Third, the evaluation agent uses the interface to play the game under the corresponding gameplay modes and record its trajectories. Finally, the game is scored from these trajectories and the final score is obtained by averaging across all instances.

G.2 HUMAN-EVALUATION QUESTIONNAIRE

Overview. This section details the human evaluation behind the results reported in section 5. The questionnaire covers the same four RPG-Bench dimensions as the main evaluation, namely requirement fidelity (RF), content richness (CR), visual and interaction quality (VI), and playability (PL). Each question asks about what the evaluator directly observed while playing, not about code-level properties or general impressions. A single successful playthrough of a branching RPG can easily miss optional content and failures that surface only after revisiting earlier scenes, so overall impressions alone are not a reliable basis for scoring.

Evaluators. Fifteen evaluators take part in the study. Before scoring, we walk every evaluator through the questionnaire and explain the four dimensions, the behaviorally anchored response options, and how to record answers.

Playtest procedure. We require each evaluator to play the game using five different methods for a comprehensive gameplay experience before answering the questionnaire. These are a normal main-route playthrough, a side-quest playthrough, attempts to access later content before satisfying the prerequisites, repeated and out-of-order interactions after returning to earlier scenes, and repeated checks of inventory, items, quests, shops, and scene transitions. Evaluators select the option that best matches what they directly observe. Intended constraints such as collision boundaries or prerequisite-gated routes are not counted as bugs when the game communicates them clearly and preserves a valid progression path. For example, a locked door whose key is signposted in dialogue is not a defect.

Scoring. Each dimension question offers four ordered, behaviorally anchored response options. A dimension score is the mean of its question scores. Each dimension score is then multiplied by the weight of a global question, which reflects the percentage of game content a player can successfully experience. This scaling incentivizes games that allow players to smoothly experience more content to achieve higher scores. The four options of the global question carry weights of 0, 0.3, 0.6, and 1.0 in increasing order of game completion. The overall score aggregates the four scaled dimension scores using the same weighted combination as the automatic evaluation in equation (5), namely $0.1S_{\mathrm{RF}} + 0.3S_{\mathrm{CR}} + 0.3S_{\mathrm{VI}} + 0.3S_{\mathrm{PL}}$.

Global Question

Global Can the game be played through successfully?

- The game cannot be entered, or it freezes right after startup.
- The game runs, but progress stops before any ending can be reached.
- The game runs, and one ending can be reached.
- The game runs, and more than one ending can be reached.

Requirement Fidelity (RF)

A1. How would you rate the game's coverage of the core content specified in the given requirement prompt?

Content Richness (CR)

B1. Does the opening clearly communicate the current situation, the protagonist's identity, the current objective, and the reason for acting?

B2. How would you rate the main storyline?

B3. How would you rate the narrative branches?

B4. Do important story events have clear causes and consequences?

B5. What is the quality of the game's dialogue, including spoken dialogue, narration, and internal monologue?
B6. How would you rate the side-quest design?
B7. How would you rate the overall length of the game flow?

Visual and Interaction Quality (VI)

C1. Is the information on the primary interfaces complete, clear, easy to read, and easy to operate?
C2. Is the information on secondary interfaces, such as the inventory and skill interfaces, complete, clear, easy to read, and easy to operate?
C3. Are NPCs clearly visible and easy to locate on the map?
C4. Are interactive objects clearly visible and easy to locate on the map?
C5. Do game scenes make the locations of entrances and exits clear to the player?
C6. Do the maps and overall visuals form a unified, attractive, and distinctive visual style?
C7. Are the main character designs attractive, easy to distinguish, and appropriate to the characters' identities and story context?
C8. Do character movement, interactions, and scene changes have clear and appropriate dynamic presentation?
C9. Is all UI text displayed completely, without overflow, clipping, or overlap?
C10. Do the visible interaction regions match the objects that can actually be interacted with?
C11. How would you rate the character portrait design?

Playability (PL)

D1. Is the player consistently aware of the next objective during play?
D2. Before using a skill in combat, can the player understand its target, cost, cooldown, and primary effect?
D3. Are combat statistics and player–enemy battle configurations reasonable and varied?
D4. Does combat provide multiple distinct skill options?
D5. Are combat animations and visual feedback attractive, and can the effects of different combat skills be clearly distinguished?
D6. After considering the complete experience, what level of completion does the game achieve?
D7. Does map collision behave appropriately while the character moves?
D8. Does map occlusion behave appropriately while the character moves?
D9. Does the character have a progression process, such as leveling, skill improvement, or attribute changes, and does this progression affect the gameplay experience?
D10. When following the prescribed routes, can the main and side quests be completed in full and produce the correct results at their conclusions?
D11. During sequence-breaking and out-of-order tests, does the game preserve correct narrative, quest, character, and reward states?
D12. After completing the prescribed functional operations and repeating them, do the game's functions, numerical values, and interface states remain correct?

H BASELINE DETAILS

H.1 BASELINE GAME-GENERATION DETAILS

For the baseline methods, each dataset instance supplies the story premise and complexity constraints as a single requirement block. The asset-generation slot is filled with the same skills as in the complexity-scaling prompts. The verification section of the prompt instructs each baseline to build the project, complete end-to-end playthroughs, and fix any defects found, with the iteration depth controlled autonomously by each harness.

Baseline Game-Generation Prompt

Purpose. Generate a baseline game directly from one dataset requirement.

```
# Role
You are an expert game designer and game-development agent specializing in complete, polished, and playable top-down 2D RPGs.
# Task
Build a top-down 2D RPG in the src directory using Vite and Phaser. Implement the complete game rather than a design document or partial prototype.
# Game Requirements
[INSERT THE DATASET INSTANCE HERE, INCLUDING ITS STORY PREMISE AND COMPLEXITY REQUIREMENTS]
# Shared Game Systems
- Implement turn-based combat with multiple purposeful skills and clear feedback.
- Implement inventory, usable items, equipment, and gameplay-relevant skill upgrades.
- Implement reliable collision and occlusion so that characters cannot cross walls, fixed objects, or map boundaries.
- Persist structured state for quests, choices, relationships, and endings as required by the dataset instance.
- Provide clear interaction prompts, task status, and feedback about important actions.
# Interaction Constraints
- Trigger scene transitions by moving through visible exits, not by selecting scene buttons.
- Complete quest objectives through interaction with the corresponding NPC or world object, never directly from the quest panel.
- Use WASD for movement and Space for interaction.
- Do not require the player to make decisions under a countdown timer.
# Asset Generation
[ASSET-GENERATION SKILLS]
# Verification
- Ensure that every chapter, scene, battle, quest stage, objective mode, state transition, and ending required by the dataset instance is reachable and updates correctly during normal play.
- Build the project with npm run build and run it with npm run start.
- Complete representative end-to-end playthroughs that cover the routes and endings required by the dataset instance.
- Fix progression blockers, interaction failures, and state errors found during testing.
- Stop every server started for this task when testing finishes.
```