QuestGuard: Quality Gates and Diversity-Aware Repair for Reliable LLM-Generated Game Quests

Lucas Vieira dos Santos Souza
Institute of Science and Technology of Sorocaba (ICTS)
São Paulo State University (UNESP)
Sorocaba, Brazil
lucas.v.souza@unesp.br

Murilo Mazzotti Silvestrini
Institute of Mathematics, Statistics and Scientific Computing (IMECC)
University of Campinas (UNICAMP)
Campinas, Brazil
m261854@dac.unicamp.br

Leopoldo Lusquino Filho
Institute of Science and Technology of Sorocaba (ICTS)
São Paulo State University (UNESP)
Sorocaba, Brazil
Recod.ai lab, Institute of Computing
University of Campinas (UNICAMP)
Campinas, Brazil
leopoldo.lusquino@unesp.br

Abstract

Once an LLM-generated quest crosses the boundary from prose into a game engine, it becomes a software artifact: it must satisfy a structural interface, link to existing world entities, expose an executable objective flow, and provide testable completion conditions. QuestGuard addresses this software-engineering problem through a modular Software Quality Assurance pipeline in which a JSON Schema acts as a lightweight type system and executable contract, deterministic validators operate as quality gates, and rejected artifacts enter a bounded repair–revalidation loop before publication. The architecture combines structural, referential, graph, and content-rule checks with an evidence-adjudicated semantic audit, and compares validity-oriented repair with a diversity-aware alternative. We report an exploratory proof-of-concept study with five configurations, three independent runs, and 90 quests per configuration. Prompt-only and schema-guided generation produced no fully valid artifacts, although schema guidance reduced the mean number of detected errors from 9.67 to 3.38 per quest. Since the repair configurations are designed to iterate correction and revalidation until artifacts pass or bounded attempts are exhausted, the main empirical result is the repair overhead required to reach deterministic validity: both repair configurations reached 100% final yield, while C4 used two and C5 used eight LLM repair calls across 90 quests. Diversity-aware repair increased mean unique structural signatures from 14.00 to 17.67, reduced duplicate signatures from 83.33% to 63.33%, and lowered entity concentration and pairwise similarity. QuestGuard therefore improves deployability under explicit software contracts; it does not claim to optimize entertainment value, and its diversity measures capture structural rather than complete experiential or narrative diversity.

Keywords

Large Language Models, Procedural Content Generation, Game Quests, Software Quality Assurance, Quality Gates, Continuous Integration, Automated Repair

1 Introduction

Large language models (LLMs) are increasingly used to generate dialogue, characters, quests, and world descriptions in games [2, 15]. Their flexibility is attractive for procedural content generation (PCG), but stateful game systems expose a central reliability problem: plausible language is not equivalent to valid game state. Prior work on GPT-based RPG quest generation shows the promise of language models for quest authoring while also reporting substantial quality variation and unusable outputs [17]. Similarly, work on AI game masters and LLM-driven NPCs reports rule violations, inconsistent state updates, hallucinated items, and references that do not belong to the current game world [5, 13]. These failures are especially costly when generated content is consumed automatically rather than reviewed as draft prose.

This paper frames quest generation as a software-engineering problem. A machine-consumable quest is a structured specification whose fields form an interface with the engine, whose identifiers must resolve against a world model, whose objectives form an executable dependency graph, and whose completion predicates must be observable. In this setting, generation resembles a nondeterministic build step: it creates a candidate artifact, but does not establish that the artifact is safe to integrate. A release discipline is therefore required between generation and publication.

QuestGuard provides that discipline. It treats the quest schema as a lightweight runtime type system and as a design-by-contract boundary between the generator and the game engine [7, 10]. It then applies a sequence of automated quality gates, conceptually aligned with Software Quality Assurance (SQA) and Continuous Integration (CI): every candidate is checked, defective artifacts are held, repairs are revalidated, and only passing artifacts are promoted [1, 3, 12]. The architecture is not merely a prompt template; it is a modular assurance pipeline with explicit contracts, repeatable analyses, evidence, and release decisions.

The evaluation compares five configurations. C1 performs prompt-only generation. C2 adds schema and design guidance. C3 applies a fail-closed publication gate to the same C2 outputs. C4 repairs rejected artifacts with validity as the primary objective. C5 uses the same acceptance criteria but selects repairs with awareness of structural signatures and entity usage. The study asks how schema guidance changes error density, how quality gates and repair affect valid-artifact yield, whether diversity-aware repair improves structural diversity, and how often repair requires an additional LLM call.

The contributions are a software-architecture view of generated quests as contract-governed artifacts; a modular validation pipeline that maps quest defects to established software analyses; a CI-like

quality-gate and repair cycle; a diversity-aware repair policy; and an exploratory comparison across 180 independently generated C1/C2 artifacts and 180 final C4/C5 outputs. The study emphasizes deployability under explicit constraints. Human judgments of fun, pacing, narrative appeal, and replay value remain outside the current optimization target.

2 Background And Software Engineering Foundations

Search-based and machine-learning-based PCG distinguish content production from the evaluation of playability, novelty, controllability, and quality [11, 14, 16]. Quality-diversity research further argues that a generator should return not only acceptable artifacts but also a varied set of solutions [4]. LLMs extend this design space, and GPT-based quest generation has already shown that language models can support RPG quest authoring, but also that generated quests vary greatly in quality and still require evaluation [17]. Surveys identify narrative and NPC generation as major application areas, while empirical systems show that explicit state controls and game-specific functions are often needed to preserve world coherence [2, 5, 13, 15].

QuestGuard interprets these risks through software-engineering abstractions. First, the JSON Schema plays the role of a lightweight dynamic type system: it defines the admissible shape of a quest, including required fields, value domains, and nested structures. It also acts as an executable contract between a supplier—the generator—and a client—the game engine. In design-by-contract terms, the generator must establish structural postconditions before the engine may rely on the artifact [7]. The contract is necessary but incomplete because many obligations depend on external world state or relationships among multiple objectives.

Second, the remaining gates correspond to familiar analysis stages. Referential validation is analogous to linking and symbol resolution: identifiers emitted by the generator must resolve in the world repository, and actions must be compatible with the resolved entity types. Graph validation resembles control-flow analysis: objective nodes and prerequisite edges must define a reachable, acyclic execution order. Content rules form a domain-specific static analysis that rejects unobservable completion predicates and other patterns known to be unsuitable for execution. These analogies do not rename the validators; they explain why the proposed pipeline belongs to software engineering rather than only to prompt engineering. Static program analysis similarly reasons about possible runtime behavior before execution [8].

Finally, the complete lifecycle follows a CI-style promotion model. Continuous Integration repeatedly subjects candidate changes to automated build and test activities before they enter a shared product state [1, 12]. In QuestGuard, generation produces the candidate build, validators implement automated checks, C3 prevents failed artifacts from being published, and C4/C5 implement correction followed by revalidation. This is a content-integration pipeline rather than a source-code pipeline, but the SQA objective is the same: prevent known defects, detect violations consistently, record evidence, and control promotion through explicit quality criteria. Quality gates are therefore not auxiliary filters; they are the release boundary of the architecture [3].

3 QuestGuard Architecture

3.1 Contracts, World Model, And Quest Representation

QuestGuard uses a typed world repository as the authoritative symbol table for the game domain. The experimental world contains 17 entities categorized as `npc`, `location`, `item`, `object`, or `enemy`. Each entity has a stable identifier and may include attributes such as name, role, or location. Quest references are expressed through these identifiers, allowing the pipeline to distinguish a valid syntactic string from a link that resolves to an actual game entity.

A quest is a JSON object containing its identifier, title, summary, type, giver, start location, structured objectives, completion conditions, rewards, tags, and design notes. Each objective contains a step identifier, action, target, dependencies, and a success condition. The schema specifies the structural contract, whereas the world repository and the objective graph provide the context required to evaluate cross-field and cross-artifact obligations.

QuestGuard does not assume a particular game engine or a fixed quest architecture. Its game-specific integration boundary is concentrated in three artifacts: the quest schema, the typed world repository, and the domain-specific content rules. Adapting the pipeline to another game therefore requires implementing these contracts and mappings for the target domain, while the validation–repair–revalidation–publication workflow remains unchanged. In this sense, the architecture is engine-independent but not game-model-independent: it applies to games whose quests can be represented as structured artifacts and whose relevant world entities, relations, and observable state predicates can be exposed to the assurance layer.

Figure 1 summarizes the end-to-end flow. A candidate is generated, analyzed, either accepted or held, repaired when appropriate, and revalidated before publication. Published sets are then evaluated for structural diversity.

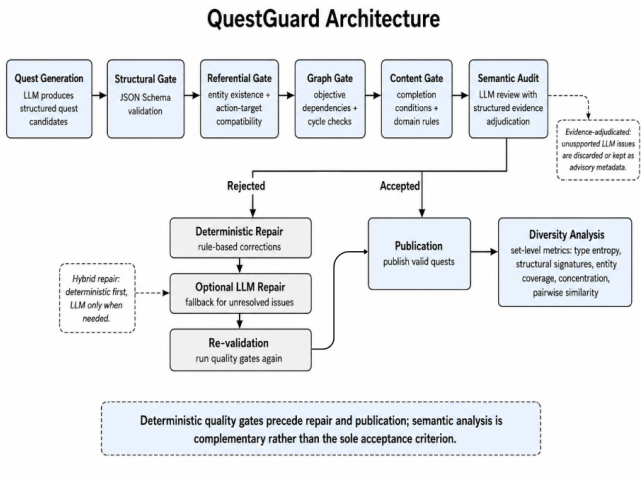

Figure 1: QuestGuard architecture. Deterministic quality gates precede repair and publication; semantic analysis is evidence-adjudicated and complementary.

3.2 Quality Gates And Continuous Assurance

The structural gate checks the JSON Schema contract and detects absent properties, invalid types, malformed nested structures, illegal enumerations, and insufficient objective counts. Passing this gate means that downstream modules receive a well-formed artifact; it does not mean that identifiers link or that the quest can execute.

The referential gate performs world linking. It resolves the giver, start location, objective targets, and reward references, then verifies action–target compatibility. For example, `visit` requires a location, `talk` an NPC, `collect` an item, `inspect` an object, and `defeat` an enemy. Unknown identifiers are unresolved symbols, while incompatible action–target pairs are type errors that become visible only after linking.

The graph gate analyzes objective control flow. Step identifiers must be unique, dependency references must resolve, the dependency graph must be acyclic, and the quest must contain executable roots and terminal objectives. A cycle represents an impossible control-flow condition; a missing dependency represents a branch to a nonexistent node.

The content-rules gate performs domain-specific static analysis over constraints not captured by structure, linking, or graph topology. In the current implementation, generic completion conditions that merely restate quest completion are treated as errors, while weak preconditions and short or underspecified objective success conditions are retained as warnings. Error-level findings prevent deterministic acceptance, whereas warning-level findings remain diagnostic. These rules are deliberately game-specific and form part of the adaptation boundary described in Section 3.1. Together, the four deterministic gates define the valid-artifact criterion used in the experiment.

The evidence-adjudicated semantic audit is a constrained diagnostic stage rather than an open-ended LLM review. This restriction responds to known limitations of LLM-as-a-judge evaluation, including bias, inconsistent reasoning, and unsupported explanations [18]. The model scores narrative consistency, gameplay clarity, integration readiness, reusability, maintainability, and game-design quality, and may propose only predefined issue categories. QuestGuard assigns severities to these findings: world inconsistencies, untracked entities, and unmodelled combat dependencies are high-severity findings; unclear objectives are warnings; and observations about reuse, maintainability, or design quality remain advisory. In the present experiment, however, semantic findings are reported as complementary diagnostic evidence and do not alter the deterministic valid-artifact criterion. A deployment policy may optionally promote adjudicated high-severity semantic findings to publication-blocking conditions. Low semantic scores alone never reject an artifact.

Each proposed issue must include verifiable evidence, such as an artifact path, an entity identifier, an objective index, and values observed in the quest. A deterministic adjudicator checks this evidence against the quest and the world repository before the finding is retained. For example, `QUEST_GIVER_STATE_CONTRADICTION` is accepted only when the same giver identifier is also used as a rescue or combat target, or when an unavailable giver is still required by a later objective. Unsupported claims, invalid indices, and evidence contradicted by the artifact are discarded rather than being treated as validation failures. The complete issue-code taxonomy and evidence rules are provided with the research artifact.

C3 makes the CI/SQA role of the gates explicit. It publishes only artifacts that pass the deterministic contract, linking, control-flow, and static-rule checks. A zero publication rate is therefore not a malfunction of C3; it means the gate prevented known defects from crossing the integration boundary. C4 and C5 extend the same cycle with automated correction, after which the complete gate suite is executed again.

3.3 Repair And Structural Diversity

The repair stage follows the same defect–correction–revalidation pattern studied in automated program repair, but applies it to generated content artifacts rather than source code [6]. Both C4 and C5 implement bounded repair loops: a candidate is modified, the deterministic gate suite is executed again, and the artifact is accepted only if the contract is satisfied. Therefore, final yield should be interpreted together with the repair overhead of reaching it, including deterministic transformations, LLM fallback calls, and any failures within the configured bounds.

C4 first applies deterministic transformations derived from validation reports: it normalizes required fields, renumbers steps, removes invalid dependencies and cycles, replaces unresolved entities, corrects incompatible actions or targets, and rewrites generic completion conditions. Only unresolved cases are sent to an LLM repair fallback. Because C4 prioritizes validity, different defects may converge to the same convenient valid pattern. For example, in the worked trace of Figure 2, an incompatible `visit:object` objective is repaired by replacing the target with a valid location.

C5 preserves the same validity contract but changes deterministic candidate selection. It prefers to retain a valid target and change the action, uses less frequent compatible entities when replacement is unavoidable, and penalizes repairs whose structural signature is already common in the accepted set. Thus, `visit:object` is preferably converted to `inspect:object`, rather than replacing the object with another location. Figure 2 shows this behavior in an actual repair trace from Run 1. The selected C2 artifact contained several schema violations and, in particular, an `INCOMPATIBLE_ACTION_TARGET` finding caused by `visit:object`. The figure focuses on this referential defect to contrast the repair decisions of C4 and C5; both strategies also normalized the remaining invalid fields before full-suite revalidation.

For a quest q, the structural signature is the ordered sequence

$$\sigma(q) = \langle a_1{:}\tau(t_1), a_2{:}\tau(t_2), \ldots, a_m{:}\tau(t_m) \rangle, \tag{1}$$

where a_i is an objective action, t_i its target, and $\tau(t_i)$ the target type resolved from the world repository. The repair state records signature and entity frequencies and is updated only after a candidate passes all deterministic gates. This notion measures structural variation in executable quest representations. It does not represent complete narrative or experiential diversity: two quests may use different entities and signatures while producing similar pacing, challenge, or player experience.

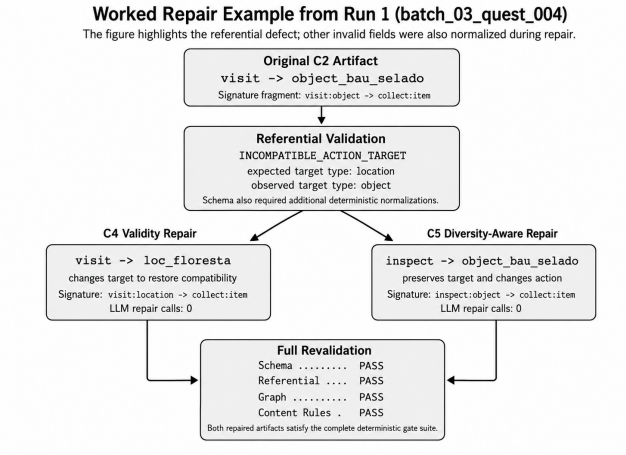

Figure 2: Worked repair example from Run 1, `batch_03_quest_004`. The highlighted C2 defect combines `visit` with an `object` target, producing an `INCOMPATIBLE_ACTION_TARGET` finding. C4 restores compatibility by replacing the target with a location, whereas C5 preserves the grounded object and changes the action to `inspect`. After the remaining deterministic normalizations, both repaired artifacts pass the complete gate suite without an LLM repair call.

4 Exploratory Study Design

4.1 Configurations And Protocol

Figure 3 presents the controlled relationship among the configurations. C1 and C2 perform independent generations. C3, C4, and C5 operate on the same C2 artifacts within each run, enabling paired comparison of gating and repair rather than comparison of unrelated samples.

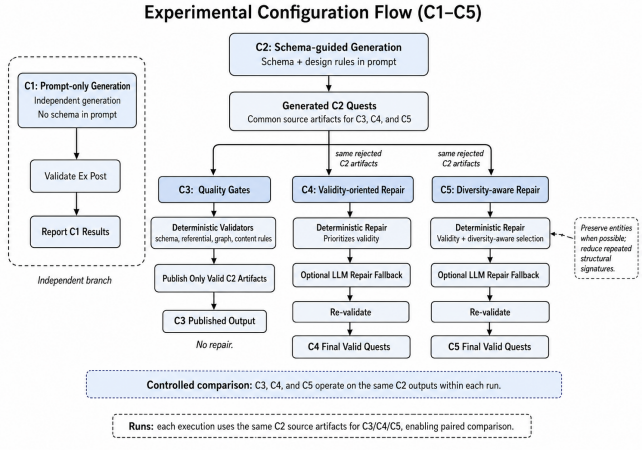

Figure 3: Controlled experimental configurations. C3, C4, and C5 operate on the same C2 outputs within each run.

All generations used the locally executed `llama3.2` model through Ollama [9]. The local environment used Ollama version 0.31.2 with 3.2B parameters, 131072-token context length, 3072-dimensional embeddings, and `Q4_K_M` quantization. Generation temperature was 0.7, review temperature was 0.1, `top_p` was 0.9, and the maximum output was 4096 tokens per call. Up to eight calls could complete a batch. When output reached the token limit, complete quest objects were recovered from the partial JSON and only the shortfall was requested again.

Each run used an automatically generated base seed, while individual request seeds were derived from the base seed and request index. This design makes runs reproducible without forcing every retry to repeat the same partial output. Raw responses, settings, seeds, validation reports, repair traces, and file hashes were preserved.

The study is exploratory and establishes a proof of concept rather than an inferential statistical claim. We use the term *proof of concept* because the experiment validates the architecture and its assurance lifecycle under one controlled quest representation, one typed world model, and one model setup; it does not demonstrate cross-game, cross-engine, or cross-model generalization. Three independent final runs were executed, each with three batches of ten quests. Across the runs, C1 produced 90 prompt-only artifacts and C2 produced 90 schema-guided artifacts. The same 90 C2 artifacts were evaluated by C3 and supplied to C4 and C5, yielding 90 final outputs for each repair strategy. Distinct seeds and file hashes confirmed that the generated runs differed. With only three runs, results are reported through means, sample standard deviations, and paired deltas; future experiments should use more runs, additional models, and formal statistical tests.

4.2 Metrics And Scope

A valid artifact is a quest that passes the deterministic structural, referential, graph, and content-rule gates. The evidence-adjudicated semantic audit is analyzed separately and does not change ValidityRate or publication yield in the present experiment. For N generated artifacts,

$$\text{ValidityRate} = \frac{N_{\text{valid}}}{N}. \tag{2}$$

For a repair configuration, final yield is

$$\text{FinalYield} = \frac{N_{\text{valid after repair}}}{N_{\text{originally generated}}}, \tag{3}$$

Because every C2 artifact required repair in the reported experiment, repair success and final yield coincide for C4 and C5; a separate repair-success-rate metric would therefore be redundant. The study also reports validation errors and warnings per quest, generation calls, and LLM repair calls. Structural diversity is measured through normalized quest-type entropy, unique structural signatures, duplicate-signature rate, entity coverage, entity concentration, and average pairwise Jaccard similarity over quest type, action–target-type pairs. Higher entropy, signature count, and coverage indicate more structural variety; lower duplication, concentration, and similarity indicate less homogenization.

These metrics deliberately optimize deployability and structural variety, not entertainment value. No human study currently measures fun, narrative interest, emotional impact, challenge, pacing, or replayability. We therefore use the term *structural diversity*, not

complete narrative diversity. A future model should combine player judgments.

5 Results

5.1 Deployability And Repair Overhead

Tables 1 and 2 summarize deterministic validity, error density, generation calls, and LLM-call repair overhead. Neither C1 nor C2 produced a fully valid quest. Schema guidance nevertheless reduced the mean error count from 9.67 to 3.38 per quest.

Table 1: Validity and validation errors across three final runs.

Configuration	Yield	Errors	Warnings
C1 Prompt-only	0.00 ± 0.00	9.67 ± 1.13	0.00 ± 0.00
C2 Schema-guided	0.00 ± 0.00	3.38 ± 1.31	0.30 ± 0.30
C3 Quality gates	0.00 ± 0.00	3.38 ± 1.31	0.30 ± 0.30
C4 Validity repair	**1.00 ± 0.00**	0.00 ± 0.00	0.00 ± 0.00
C5 Diversity-aware	**1.00 ± 0.00**	0.00 ± 0.00	0.00 ± 0.00

Table 2: Generation calls and LLM-call repair overhead across three final runs.

Configuration	Generation calls	LLM repair calls
C1 Prompt-only	3.33 ± 0.58	–
C2 Schema-guided	5.67 ± 0.58	–
C3 Quality gates	5.67 ± 0.58	–
C4 Validity repair	5.67 ± 0.58	0.67 ± 1.15
C5 Diversity-aware	5.67 ± 0.58	2.67 ± 1.53

The relative reduction in error density from C1 to C2 was

$$\frac{9.67 - 3.38}{9.67} \times 100 \approx 65.1\%. \tag{4}$$

The contract supplied in C2 therefore improved artifact shape without establishing deployability. C3 correctly withheld all 90 C2 artifacts, demonstrating the fail-closed behavior expected from an integration gate. C4 and C5 then executed bounded repair–revalidation loops and recovered the complete set in all three runs. The 100% deterministic final yield should therefore be read as evidence that the implemented repair rules and fallback bounds covered the observed defect classes; the more informative result is the repair overhead required to reach this yield. Figure 4 presents these results.

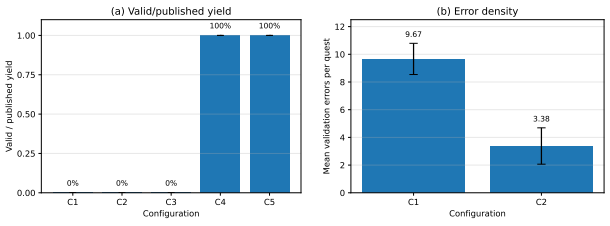

Figure 4: Validity and error density across configurations.

C4 used two total LLM repair calls across 90 quests and C5 used eight, corresponding to approximately 0.022 and 0.089 calls per original quest. This low fallback rate indicates that most corrections were deterministic. Diversity-aware selection required slightly more fallback assistance, but the additional LLM-call overhead remained small relative to the generated set. We do not equate call count with wall-clock latency, token consumption, or monetary cost.

5.2 Gate-Level Defect Distribution

To determine which assurance layers contributed to defect detection, we decomposed the validation reports of the same 90 C2 artifacts by deterministic gate. Table 3 reports both the number of artifacts for which each gate produced at least one finding and the total number of errors and warnings. The final column shows how many artifacts independently passed each gate; it is a diagnostic single-gate analysis rather than a separate repair experiment.

The structural gate detected by far the largest number of violations, with 263 errors affecting 89 of the 90 artifacts. Referential validation detected 32 errors across 25 artifacts, while graph validation found nine errors across nine artifacts. The content-rules gate produced 27 warnings across 18 artifacts but no error-level findings in these runs. Importantly, no single gate subsumed the complete assurance suite. The only C2 artifact that passed the structural gate still failed referential validation because a reward referenced an entity absent from the world repository. This provides a concrete example of why structural conformance alone does not establish deployability.

Table 3: Defect distribution by deterministic quality gate over the 90 C2 artifacts.

Gate	Affected	Errors	Warnings	Pass
Structural	89	263	0	1
Referential	25	32	0	65
Graph	9	9	0	81
Content Rules	18	0	27	90

5.3 Structural Diversity

Tables 4 and 5 compare C4 and C5. Quest-type entropy was identical because repair preserved the original quest types.

Table 4: Structural diversity across three final runs.

Metric	C4	C5	Δ
Quest-type entropy	0.885 (0.029)	0.885 (0.029)	0.000 (0.000)
Unique signatures	14.00 (3.00)	**17.67 (6.35)**	**+3.67 (4.04)**
Duplicate rate	0.833 (0.088)	**0.633 (0.260)**	**−0.200 (0.219)**

Relative to C4, C5 increased unique signatures by approximately 26.2%, reduced duplicate signatures by 24.0%, reduced entity concentration by 14.2%, and reduced pairwise similarity by 15.1%. Run-level deltas were nonnegative for unique signatures, nonpositive for duplicate rate, and negative for concentration and similarity in all runs.

Table 5: Entity-use and similarity metrics across runs.

Metric	C4	C5	Δ
Entity coverage	0.843 (0.034)	**0.863 (0.034)**	**+0.020 (0.034)**
Entity concentration	0.150 (0.016)	**0.128 (0.005)**	**−0.021 (0.015)**
Pairwise similarity	0.155 (0.017)	**0.132 (0.010)**	**−0.023 (0.012)**

5.4 Evidence-Adjudicated Semantic Audit

The evidence-adjudicated semantic audit was executed over the 90 final C5 artifacts. All evaluations completed successfully in a single LLM attempt. Across the three runs, the LLM proposed 169 candidate semantic findings: 56, 56, and 57, respectively. None survived deterministic evidence adjudication, so all 169 were discarded rather than promoted to validated semantic findings.

Most rejected findings were contradicted by the quest fields (84/169, 49.7%) or concerned objectives that the adjudicator determined to be concrete (73/169, 43.2%). Six findings relied on unsupported evidence rules, five referenced invalid objective indices, and one claimed an untracked entity that was present in the world repository. The occasional malformed objective index also illustrates why model-produced evidence is verified rather than trusted directly.

These results, presented in Table 6, do not establish that the semantic audit has perfect recall or that the artifacts contain no semantic defects. Instead, they demonstrate the role of the adjudication layer: candidate LLM judgments do not affect the assurance decision unless their supporting evidence can be verified against the artifact and world model.

Table 6: Evidence-adjudicated semantic audit across the three final runs.

Run	Proposed	Accepted	Discarded
Run 1	56	0	56
Run 2	56	0	56
Run 3	57	0	57
Total	**169**	**0**	**169**

Taken together, the results show that schema guidance substantially reduced error density but did not establish deployability; the deterministic gates contributed complementary forms of assurance; bounded repair recovered all observed invalid artifacts; and diversity-aware repair improved structural variety while preserving the same validity contract. The semantic audit further showed that LLM-proposed concerns require independent evidence checking: none of the 169 candidate findings in the final C5 sets survived deterministic adjudication. Only a small number of additional LLM repair calls were required by C4 and C5.

6 Discussion And Threats To Validity

6.1 Software Engineering Implications

The comparison between C2 and C3 illustrates why a generated contract is not equivalent to an assured artifact. The schema acts as a type and interface boundary, but a well-typed quest can still fail linking, control-flow, or domain-static-analysis checks. This mirrors conventional software pipelines, in which successful parsing or compilation does not replace integration testing and quality assurance.

A natural alternative is constrained decoding. Ollama supports structured outputs by passing a JSON Schema to the `format` parameter, which can force the response to match a schema more reliably than prompt-only guidance [9]. This mechanism is complementary to QuestGuard rather than a replacement for it. It can reduce or eliminate many format errors, but it does not prove that identifiers exist in the game world, that actions match resolved target types, that objective dependencies are acyclic, or that completion conditions are observable by the engine. It also does not remove the practical need to recover usable objects when long generations are truncated by output limits. For this reason, QuestGuard focuses on the assurance layers that grammar-constrained decoding does not cover.

C3 is particularly important from a CI perspective. Its publication yield was zero because all C2 artifacts violated at least one deterministic obligation. In a generation-only evaluation, this result might appear negative; in SQA terms, it is the intended behavior of a quality gate. The pipeline prevented defects from being promoted and preserved evidence for repair. C4 and C5 then completed a closed assurance loop: detect, correct, reanalyze, and publish. This separation also improves maintainability because each gate has a narrow responsibility and can evolve independently as schemas, worlds, or domain rules change.

The results further suggest that deterministic repair is a practical complement to LLM generation. Most defects were corrected without another model call, making the process more reproducible and auditable. The LLM fallback remains useful for coordinated rewriting, but it is not the sole assurance mechanism.

6.2 Deployability, Experiential Quality, And Study Limitations

QuestGuard optimizes deployability rather than entertainment value. Passing the gates means that the artifact satisfies the defined contract, resolves against the world, contains executable objective flow, and uses observable domain conditions. It does not establish that the quest is fun, emotionally engaging, well paced, balanced, or worth replaying. This is the principal construct-validity limitation of the current study and should not be hidden behind the 100% valid-artifact yield.

The same distinction applies to diversity. The metrics combine action–type signatures, entity usage, entropy, and pairwise similarity; they measure structural diversity in the quest representation. They do not measure experiential diversity. Distinct signatures may still lead to similar gameplay loops, while structurally similar quests may feel different because of narrative context, challenge, timing, or player choice. Future work should therefore build an experiential-diversity layer using human evaluations and cost functions for fun, novelty, pacing, challenge, replayability, and perceived gameplay difference.

The semantic audit partially addresses qualities not encoded by deterministic validators, but it is not human ground truth. Evidence adjudication limits unsupported model judgments, yet aesthetic and experiential claims remain subjective. A blinded human study

involving players and designers is required to calibrate semantic scores and determine whether structurally diverse outputs are also perceived as narratively and experientially diverse.

The study is exploratory. It uses one local LLM, one small RPG world, one schema, 17 entities, and three runs. The shared C2 artifacts support paired comparisons among C3–C5, and stored seeds, responses, and hashes improve reproducibility, but the sample is not sufficient for strong statistical generalization. Accordingly, the architectural mechanism is intended to be model-independent, whereas the observed defect distribution, repair overhead, and magnitude of the C5 diversity gains are empirical properties of this particular setup and should not be generalized to other models or worlds without replication. The high C4 duplicate rate may also be influenced by the small action and entity space. The next evaluation should increase the number of independent runs, test multiple model families and world sizes, report confidence intervals, and apply paired statistical tests and effect sizes. Engine-level execution tests are also needed to move from contract satisfaction toward demonstrated runtime integration.

7 Conclusion

QuestGuard reframes LLM quest generation as a software-quality problem. A generated quest is treated as a contract-governed software artifact, the JSON Schema serves as a lightweight type and interface system, entity validation performs linking, objective-graph validation performs control-flow analysis, and content rules provide domain-specific static analysis. These checks operate as CI-style quality gates inside an SQA cycle that holds defective artifacts, repairs them, revalidates them, and promotes only passing outputs.

In the exploratory study, schema guidance reduced mean error density by approximately 65.1% but did not produce fully valid artifacts. The fail-closed C3 gate prevented all defective C2 outputs from publication. C4 and C5 then reached 100% deterministic final yield within the configured repair bounds, so the central empirical finding is not the yield alone but the low LLM-call overhead required to reach it: most repairs were deterministic, with only two LLM repair calls for C4 and eight for C5 across 90 quests. C5 preserved full deterministic validity while improving the measured structural diversity of the resulting quest sets. These results support a modular architecture in which LLMs propose content but do not define correctness.

These findings establish a proof of concept for deployability-oriented assurance, not a complete evaluation of game quality or evidence of universal game-engine applicability. QuestGuard separates an engine-independent assurance workflow from game-specific contracts, world mappings, and domain rules, but the empirical results reported here remain specific to the evaluated model and world. Future work will expand the experiment for rigorous statistical analysis, evaluate multiple LLMs and larger worlds, execute quests inside a game engine, calibrate the semantic audit with human raters, and develop experiential-diversity objectives based on fun, pacing, challenge, narrative appeal, and replayability.

Artifact Availability

The QuestGuard research artifact is publicly available at:

https://github.com/luscaassz/questguard

The repository contains the source code, JSON Schema, typed world model, generation prompts, validation and repair modules, experimental scripts, configuration files, raw LLM responses, run seeds, validation reports, repair traces, generated quest sets, and aggregated results used in this study. It also provides installation instructions and commands for reproducing the C1–C5 configurations, executing the deterministic and semantic validation pipelines, and aggregating independent experimental runs.

References

[1] Paul M. Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional.

[2] Roberto Gallotta, Graham Todd, Marvin Zammit, Sam Earle, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2024. Large Language Models and Games: A Survey and Roadmap. arXiv:2402.18659 [cs.AI] doi:10.48550/arXiv.2402.18659

[3] Marko Gluhak and Luka Pavlič. 2022. A Quality Gate Role in a Software Delivery Pipeline. In *Proceedings of the Ninth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (CEUR Workshop Proceedings, Vol. 3237)*. 1–12. https://ceur-ws.org/Vol-3237/paper-glu.pdf

[4] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2019. Procedural Content Generation through Quality Diversity. In *2019 IEEE Conference on Games*. 1–8. doi:10.1109/CIG.2019.8848053

[5] Minkyung Kim, Junsik Kim, Hwidong Bae, Woongcheol Yang, Sangdon Park, and Sohee Bae. 2025. State-Inference-Based Prompting for Natural Language Trading with Game NPCs. arXiv:2507.07203 [cs.AI] doi:10.48550/arXiv.2507.07203

[6] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. doi:10.1109/TSE.2011.104

[7] Bertrand Meyer. 1992. Applying "Design by Contract". *Computer* 25, 10 (1992), 40–51. doi:10.1109/2.161279

[8] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer. doi:10.1007/978-3-662-03811-6

[9] Ollama. 2026. Ollama Generate API Documentation. https://docs.ollama.com/api/generate.

[10] Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

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

[12] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uira Kulesza. 2022. The Effects of Continuous Integration on Software Development: A Systematic Literature Review. *Empirical Software Engineering* 27, 3 (2022), 78. doi:10.1007/s10664-021-10114-1

[13] Jaewoo Song, Andrew Zhu, and Chris Callison-Burch. 2024. You Have Thirteen Hours in Which to Solve the Labyrinth: Enhancing AI Game Masters with Function Calling. arXiv:2409.06949 [cs.CL] doi:10.48550/arXiv.2409.06949

[14] Adam Summerville, Sam Snodgrass, Michael Mateas, Santiago Ontañón, John R. H. Mariño, and Julian Togelius. 2018. Procedural Content Generation via Machine Learning. *IEEE Transactions on Games* 10, 3 (2018), 257–270. doi:10.1109/TG.2018.2846639

[15] Penny Sweetser. 2024. Large Language Models and Video Games: A Preliminary Scoping Review. In *Proceedings of the 6th ACM Conference on Conversational User Interfaces*. 1–8. doi:10.1145/3640794.3665582

[16] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186. doi:10.1109/TCIAIG.2011.2148116

[17] Susanna Värtinen, Perttu Hämäläinen, and Christian Guckelsberger. 2024. Generating Role-Playing Game Quests With GPT Language Models. *IEEE Transactions on Games* 16, 1 (2024), 127–139. doi:10.1109/TG.2022.3228480

[18] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, Vol. 36. https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html