AutoUE: Automated Generation of 3D Games in Unreal Engine via Multi-Agent Systems

Anonymous ACL submission

Abstract

Automatically generating 3D games in commercial game engines remains a non-trivial challenge, as it involves complex engine-related workflows for generating assets such as scenes, blueprints, and code. To address this challenge, we propose a novel multi-agent system, **AutoUE**, which coordinates multiple agents to end-to-end generate 3D games, covering model retrieval, scene generation, gameplay and interaction code synthesis, and automated game testing for evaluation. In order to mitigate tool-use hallucinations in LLMs, we introduce a retrieval-augmented generation mechanism that grounds agents with relevant UE tool documentation. Additionally, we incorporate game design patterns and engine constraints into the code generation process to ensure the generation of correct and robust code. Furthermore, we design an automated play-testing pipeline that generates and executes runtime test commands, enabling systematic evaluation of dynamic behaviors. Finally, we construct a game generation dataset and conduct a series of experiments that demonstrate AutoUE's ability to generate 3D games end-to-end, and validate the effectiveness of these designs.[1]

1 Introduction

Digital games, as one of the most influential forms of interactive media, play an essential role in entertainment and simulation (Martins de Freitas Cintra et al., 2025; Park et al., 2023). Due to the need to design complex visual environments and system logic, the game creation remains a highly creative yet labor-intensive process.

Recent advances in Large Language Models (LLMs) (Guo et al., 2025) and generative models (Ho et al., 2020) have motivated studies on automated game generation from natural language descriptions (Che et al., 2025; Valevski et al., 2025; Dai et al., 2024), aiming to reduce development barriers. A promising direction is to integrate LLMs with game engines to generate in-engine assets (Maleki and Zhao, 2024; Buongiorno et al., 2024), which offers better controllability, editability, and practical value for game development.

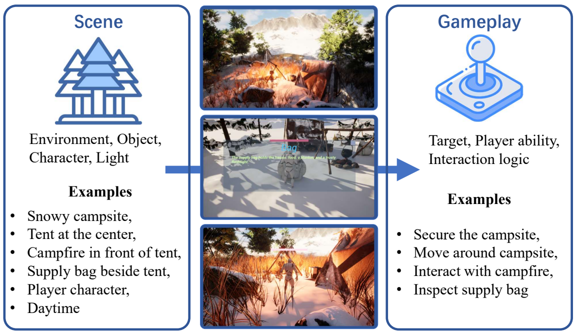


Figure 1: An example of game scene and gameplay.

Despite these advances, automatically generating 3D games in game engines remains a non-trivial challenge (Earle et al., 2025). Compared with 2D or simple game settings (Todd et al., 2024; Hu et al., 2024; Sudhakaran et al., 2023), 3D games require more complex spatial layouts, gameplay behaviors, and coordination across multiple subsystems. Recent works have integrated LLMs with commercial engines such as Unreal Engine (UE) and Unity to generate 3D scenes (SongTang et al., 2025), gameplay logic (Wayama et al., 2022), and engine code (Yang et al., 2025). However, most of these efforts focus on isolated individual components, making it difficult to generate a complete, playable 3D game end-to-end.

To address these challenges, we propose **AutoUE**, a novel multi-agent system that coordinates agents for model retrieval, scene generation, gameplay code, interactive object, and automated play-testing to produce complete 3D games in UE.

Specifically, the model retrieval agent builds an embedding database over 858K 3D models, enabling efficient semantic retrieval of relevant model assets. Based on the retrieved models, the scene

[1] The code will be open-sourced after the official publication.

generation agent uses UE's built-in tool Procedural Content Generation (PCG) to generate a visually editable layout graph that produces a coherent 3D scene. Then, the gameplay code agent follows common game-development design patterns and respects engine-specific constraints to generate modular gameplay C++ code. Building on these modules, the interactive object agent derives object interaction flows and implements the corresponding C++ code, enabling in-scene entities to provide player interactions consistent with the description. Finally, the automated play-testing agent automatically generates and executes runtime test commands to evaluate the scene and gameplay behaviors. These agents communicate via structured specifications to convey contextual information and ensure consistency across the system.

Our system has the following innovations:

(1) Compared to other engines, UE provides comprehensive built-in tools for efficient game development. To mitigate tool-use hallucinations by LLMs, we propose a retrieval-augmented generation (RAG) based mechanism that retrieves relevant information from tool documents and provides the agent with accurate usage instructions.

(2) Existing methods often overlook design patterns in game development, resulting in unconstrained code that is difficult to maintain and extend. To address this limitation, we generate functional gameplay module code and design a module management framework based on common practices. This enables the system to produce reusable and composable gameplay code, which can be invoked by interactive objects in a controlled manner.

(3) Prior work evaluates generated assets via human inspection or static checks. However, a complete game is defined by dynamic gameplay and interaction behaviors, making objective evaluation without human effort challenging. We introduce an automated play-testing pipeline that generates test commands and implements a Model Context Protocol (MCP) to execute them at runtime, enabling automated and systematic evaluation.

Finally, we construct a dataset of 20 game-generation tasks and demonstrate that our method produces higher-quality 3D scenes than existing approaches. Meanwhile, we design multiple metrics that evaluate dynamic gameplay behaviors based on the generated gameplay and interaction code as well as runtime logs, providing a benchmark for end-to-end 3D game generation in game engines.

Overall, our contributions are summarized as:

- We present a practical end-to-end multi-agent system for generating 3D games in UE. The system collaboratively constructs a coherent 3D scene, synthesizes robust gameplay and interaction code, and performs automated play-testing.
- We design a RAG mechanism that provides agents with tool-usage guidance, thereby reducing hallucinations for LLMs. During code generation, we incorporate game design patterns and engine constraints to improve the maintainability and extensibility of the generated code.
- We construct a game-generation dataset and establish an automated play-testing pipeline, providing a benchmark for 3D game generation. It offers a solid foundation for multi-agent game generation, and the overall system provides an extensible infrastructure for future work.

2 Related Work

Existing research on applying agents to games can be divided into two types:

2.1 LLMs as Game Playing Agents

One line of work treats LLMs as game playing agents and uses games as testbeds to improve the agents's ability to process complex observations. Early studies primarily focus on text-based games: Hausknecht et al. (2020) proposes an evaluation framework for interactive fiction and adopts a template-based action space to reduce complexity. Recent studies have further applied LLMs to open-world 3D games. DEPS (Wang et al., 2023) conditions planning on environment feedback and sub-goal selection to robustly complete diverse Minecraft tasks. Voyager (Wang et al., 2024) combines an expanding skill library with a feedback-driven iterative loop to achieve open-ended exploration in Minecraft. In addition, recent work has also explored agents in a wide range of game genres, including competitive (Wang et al., 2025) and action (Chen et al., 2024) games. The latest advances (Tan et al., 2025) train foundation models on large-scale gameplay data, enabling generalization to unseen games.

2.2 LLMs as Game Generation Agents

Another line of work uses LLMs to generate game content, including text/video-based and engine-integrated game generation. The former generates game experiences in the form of text narratives (Côté et al., 2018), scripted interactions (Sun et al., 2023), or video-like content (Bruce et al.,

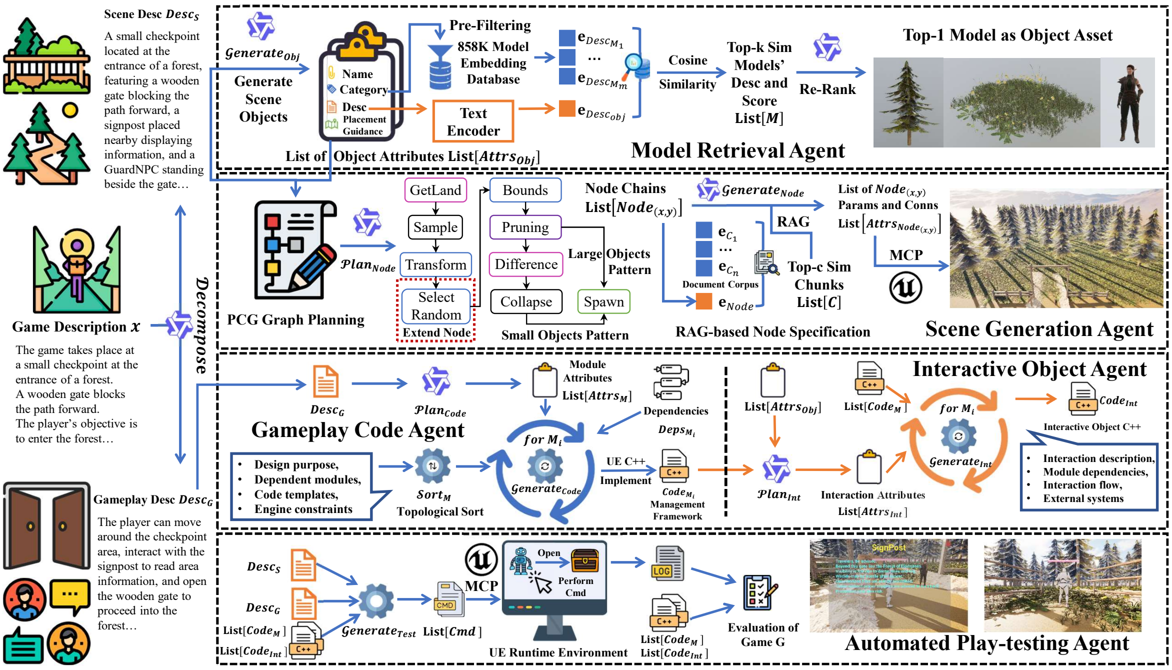


Figure 2: The overall framework of our proposed multi-agent system AutoUE. Blue arrows indicate the workflow.

2024). Such games are easier to synthesize end-to-end, but they lack free-form spatial interactions and developer-editable content. Therefore, recent studies couple LLMs with game engines to generate in-engine assets for modern game development. Sketch2Scene (Xu et al., 2024) uses diffusion models to turn user sketches into isometric 2D scene guides, and feeds guides into procedural generation to build 3D game scenes. 3Dify (Hayashi et al., 2025) uses RAG and MCP to invoke tools like Blender, Unity, and UE to create 3D scenes with iterative user feedback. UnrealLLM (Song-Tang et al., 2025) achieves automated high-quality 3D scene generation in UE by leveraging multi-modal asset retrieval and PCG powered by expert knowledge. Hassan (2025) parse game design documents and fine-tune LLMs to generate modular, compilable, and design-consistent Unity C# game template prototypes. UniGen (Yang et al., 2025) builds a multi-agent framework to generate Unity C# code, attach the code to components, and support iterative debugging. These studies focus on a single stage of the pipeline in isolation. Due to the tightly coupled nature of game engine workflows, these components are often not trivially composable. DreamGarden (Earle et al., 2025) takes a step toward end-to-end generation by using hierarchical planning and iteratively construct a game prototype in UE. Nevertheless, its generated results remain limited in overall quality and lack robustness when facing complex requirements, highlighting the need for a more reliable, workflow-aligned system for complete 3D game generation in game engines.

3 Methodology

3.1 System Overview

Figure 2 illustrates our multi-agent system, which consists of five LLMs agents: (1) Model Retrieval Agent, (2) Scene Generation Agent, (3) Gameplay Code Agent, (4) Interactive Object Agent, and (5) Automated Play-testing Agent.

We adopt the model context protocol as a standardized interface between our agents and UE. When an agent produces structured specifications that need to be integrated into UE, it invokes the engine tools or code exposed via our MCP implementation to import the outputs into the project and execute the corresponding application workflow. As MCP primarily serves as an engineering integration layer, we do not further elaborate on its implementation details in the following sections.

3.2 Problem Formulation

Given a game description x, our goal is to construct a complete UE game G that consists of a coherent 3D scene, executable gameplay code, interactive object logic, and automated testing instructions.

Since x often interleaves scene content with gameplay intent, it blurs the line between what should be built and what should happen, potentially confusing downstream agents. Therefore, we first decompose x into a scene description $Desc_S$

and a gameplay description $Desc_G$:

$$Desc_S, Desc_G = \mathcal{D}ecompose(x), \quad (1)$$

where $Desc_S$ defines the scene context and key objects, while $Desc_G$ defines the gameplay mechanics and player interactions; and $\mathcal{D}ecompose$ denotes the prompt used to query LLMs. The prompt details are provided in Appendix Figure 7.

3.3 Model Retrieval Agent

As the fundamental entities in the game world, scene objects are not only crucial for the visual presentation, but also provide the essential basis for interaction modeling. Based on $Desc_S$, we construct a list of key scene object along with their associated attributes, represented as:

$$\text{List}[Attrs_{Obj}] = \mathcal{G}enerate_{Obj}(Desc_S), \quad (2)$$

where the object attributes $Attrs_{Obj}$ contain name, category, description, and spatial placement guidance; and the prompt details of $\mathcal{G}enerate_{Obj}$ are provided in Appendix Figure 8.

The $Attrs_{Obj}$ are then used in our retrieval pipeline, which is built on TexVerse (Zhang et al., 2025), a repository with over 858K text-described 3D models from Sketchfab. To efficiently retrieve relevant models, we use a text encoder to create an embedding database from the model descriptions.

Due to the large number of models, we first apply a pre-filtering process using Sketchfab's 18 categories to narrow down the search according to the category in $Attrs_{Obj}$. Next, we calculate the cosine similarity between the encoded object description in $Attrs_{Obj}$ and the embedding of filtered models to retrieve the top-k models, formulated as:

$$\text{List}[M] = \mathcal{S}ort\left(\left[\mathcal{S}im(\mathbf{e}_{Desc_{Obj}}, \mathbf{e}_{Desc_{M_i}})\right]_{i=1}^{m}\right)[:k], \quad (3)$$

where List$[M]$ is the list of model description and similarity score of the top-k retrieved models, $\mathcal{S}ort$ denotes the function that sorts values in descending order, $\mathcal{S}im$ denotes the cosine similarity function, $\mathbf{e}_{Desc_{Obj}}$ and $\mathbf{e}_{Desc_{M_i}}$ are the embeddings of the object description and the i-th model description, and m is the number of models after pre-filtering.

Then, we use LLMs to re-rank List$[M]$ based on $Attrs_{Obj}$. The top-1 model after reranking is used as the retrieved 3D model for the object.

3.4 Scene Generation Agent

After retrieving high-quality 3D object models, we stably build a coherent UE scene based on these assets. Instead of generating hard-coded coordinates, we leverage PCG to synthesize the scene layout. PCG represents scene construction as a visually configurable graph, connecting modular nodes that support functions such as surface sampling, point transformation, and bounds modification. These nodes make PCG a natural fit for $Desc_S$, as they fulfill most of the abstract placement intentions and can be extended to meet additional needs.

3.4.1 PCG graph planning

Based on $Desc_S$ and List$[Attrs_{Obj}]$, we generate one PCG node chain per scene object. Each chain is described by an ordered list of node types with explicit editor coordinates, denoted as:

$$\text{List}[Node_{(x,y)}] = \mathcal{P}lan_{Node}(Desc_S, Attrs_{Obj}), \quad (4)$$

where List$[Node(x, y)]$ denotes a PCG node chain, whose each element is a node of type $Node$ located at editor coordinates (x, y), and the prompt details of $\mathcal{P}lan_{Node}$ are provided in Appendix Figure 9.

In practice, unconstrained PCG graph generation leads to unstable or hard-to-debug procedural behaviors. Therefore, we classify the object placement into two canonical PCG patterns, each capturing a frequent production need:

- Large objects (e.g., buildings) require stable sampling and spacing. The chain ends with a direct spawn step after pruning and bounds control to avoid overlaps and out-of-bound placements.
- Small scatter objects (e.g., trees, rocks) must avoid intersecting major actors. We add exclusion nodes to subtract forbidden regions, improving collision avoidance and visual coherence.

This design enables production-level stability, provides flexibility through parameters (e.g., density, bounds), and can meet a broader range of requirements through extended modes.

3.4.2 RAG-based node specification

The planned list only specifies which nodes appear, but it lacks the parameters and pin connections. To avoid hallucinating these details, we adopt RAG over the full document corpus: we segment all available documents into a unified set of text chunks and retrieve the most relevant chunks for each target node. In practice, source material may include cross-references across documents and sections. A strictly indexing scheme is expensive to maintain, scales poorly with new nodes, and may lead to excessive context during generation. Therefore, we use semantic retrieval to fetch the minimal guidance needed for each node, represented as:

$$\text{List}(C) = \mathcal{S}ort\left([\mathcal{S}im(\mathbf{e}_{Node}, \mathbf{e}_{C_i})]_{i=1}^{n}\right)[:c], \quad (5)$$

$$Attrs_{Node} = \mathcal{G}enerate_{Node}(Node_{(x,y)}, \text{List}(C)), \quad (6)$$

where n is the number of chunks, C_i is the i-th chunk, $\mathbf{e}_{Node}$ and $\mathbf{e}_{C_i}$ are the embeddings of $Node_{(x,y)}$ and C_i, List(C) is the retrieved top-c chunks, $Attrs_{Node}$ are the parameters and connections for $Node_{(x,y)}$, and the details of $\mathcal{G}enerate_{Node}$ are provided in Appendix Fig 10.

The RAG knowledge enables PCG to generate reliable scenes, while also making it easy to introduce new node capabilities by adding document. Finally, we develop MCP tools to convert all $Attrs_{Node}$ into the UE PCG editor. This approach can also be extended to other game development tools, as they typically modular in design and provide the necessary usage information in documents.

3.5 Gameplay Code Agent

To realize executable gameplay logic, we follow a planned generation pipeline, where the system first analyzes $Desc_G$ to plan the required functional logic modules, and then generates the corresponding UE C++ implementations, formulated as:

$$\text{List}[Attrs_M] = \mathcal{P}lan_{Code}(Desc_G), \quad (7)$$

$$\text{for } M_i \in \mathcal{S}ort_M(\text{List}[Attrs_M]) : Code_{M_i} = \mathcal{G}enerate_{Code}(Desc_G, \text{List}[Attrs_M], Deps_{M_i}), \quad (8)$$

where List$[Attrs_M]$ is the planned list of module attributes, M_i is the i-th module, $\mathcal{S}ort_M$ performs a topological sort based on the dependencies between modules, $Code_{M_i}$ is the C++ implementation code of M_i, $Deps_{M_i}$ are the generated code of modules that M_i depends on, and the details of $\mathcal{P}lan_{Code}$ and $\mathcal{G}enerate_{Code}$ are provided in Appendix Figure 11 and Figure 12, respectively.

Specifically, $Attrs_M$ contains the design purpose, usage, and dependent modules to provide more context for code generation. We also provide code templates and engine-related constraints (e.g., export/registration macros, include whitelist, logging, and cross-module access) in $\mathcal{G}enerate_{Code}$ to improve code generation stability. Additionally, following common game development practices, we design a generic module management framework as MCP for the module code. Together, these designs ensure that gameplay logic can be reliably triggered by in-engine interactions while remaining extensible and consistent with UE.

3.6 Interactive Object Agent

The generated modules (e.g., InventoryModule, DialogueModule) are invoked within the object interaction implementation methods to execute concrete gameplay logic (e.g., pickup, dialogue). To implement the interaction, we first infer which modules should be invoked and the specific order in which their methods should be called, represented as:

$$\text{List}[Attrs_{Int}] = \mathcal{P}lan_{Int}(\text{List}[Attrs_{Obj}], \text{List}[Code_M]), \quad (9)$$

where $Attrs_{Int}$ denotes the attributes of the interactive object, including interaction description, module dependencies, interaction flow, and whether external systems such as combat or navigation plugins need to be invoked; and the prompt details of $\mathcal{P}lan_{Int}$ are provided in Appendix Figure 13.

Based on these inferred attributes, we generate the C++ code for each interactive object:

$$Code_{Int} = \mathcal{G}enerate_{Int}(Attrs_{Int}, \text{List}[Code_M]), \quad (10)$$

where $Code_{Int}$ denotes the generated C++ code for the interactive object, and $\mathcal{G}enerate_{Int}$ ensures consistent interaction logic through the module's context and the interaction flow. The prompt details of $\mathcal{G}enerate_{Int}$ are provided in Appendix Fig 14.

3.7 Automated Play-testing Agent

After generating the interaction code, we have completed the entire UE game construction. However, there are multiple ways to implement $Desc_S$ and $Desc_G$, and evaluating the generated content becomes a challenge. To address this, we set up an evaluation environment by generating and executing test instructions, denoted as:

$$\text{List}[Cmd] = \mathcal{G}enerate_{Test}(Desc_S, Desc_G, \text{List}[Code_M], \text{List}[Code_{Int}]), \quad (11)$$

where List$[Cmd]$ is the testing commands used to evaluate our game G, and the prompt details of $\mathcal{G}enerate_{Test}$ are provided in Appenidx Figure 15.

There are two types of testing commands: moving to an object and performing specific interactions. We automatically execute these commands in UE runtime environment through MCP. The runtime screenshots, execution logs, List$[Node_{(x,y)}]$, List$[Code_M]$, List$[Code_{Int}]$, and middle specifications are used for the final evaluation of our generated game G, providing a thorough evaluation.

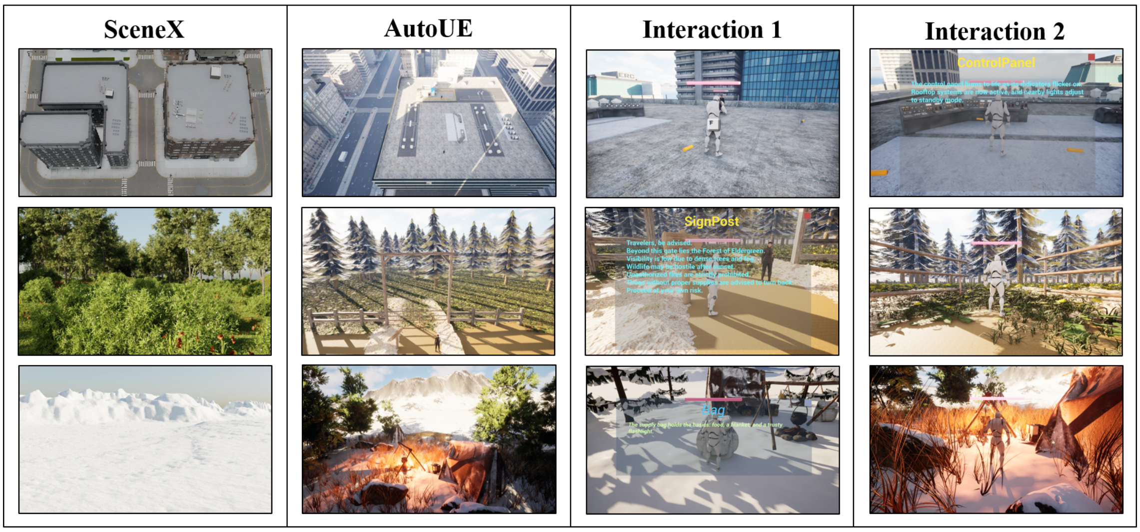


Figure 3: Visual comparison with SceneX. The first column shows scenes from SceneX; the second column shows the corresponding scenes in AutoUE; the third and fourth columns present AutoUE's interactions within each scene.

Score	Scene	Gameplay	Visual	Game
Easy	10.0	8.30	7.70	8.72
Medium	9.79	8.29	8.14	8.74
Hard	9.94	8.19	7.50	8.59
All	9.90	8.25	7.78	8.68

Table 1: The game evaluation results on PlayGen-20.

4 Experiments

We evaluate our multi-agent game generation system **AutoUE** from five perspectives, corresponding to the key claims made in previous sections:

- **Q1**: Can AutoUE generate complete, playable UE games end-to-end, and what is the overall quality of the generated games?
- **Q2**: Compared with existing scene generation baselines, are the scenes generated by AutoUE better in terms of aesthetic quality?
- **Q3**: What is the impact of RAG-based node specification on generating valid PCG graphs.
- **Q4**: What is the impact of game design patterns on the quality of the generated gameplay code?

4.1 Experimental Setup

4.1.1 Dataset

We construct a benchmark of 20 game generation tasks (denoted as PlayGen-20). These tasks cover common interactions in game design (e.g., pickup, open, dialogue, inspect, and combat). Task details are provided in the Appendix Figure 4, 5, and 6.

To better evaluate AutoUE under different levels of game design complexity, we further divide PlayGen-20 into three difficulty categories based on scene complexity and interaction complexity:

- **Easy (5 games).** These tasks focus on onboarding and practicing basic interactions. The designs include a single or low-complexity interaction, clear feedback with low failure cost. This category is well-suited for evaluating end-to-end playability and basic interaction correctness.
- **Medium (7 games).** These tasks focus on more exploration and collection. The designs include multiple interactive objects, and a slower pace that encourages players to interact more broadly with the environment. This category is well-suited for evaluating the ability to organize diverse interactive objects, enrich scene content, and achieve broader interaction coverage.
- **Hard (8 games).** These tasks introduce explicit progression constraints and stronger interaction dependencies. The designs feature clearer state transitions and tighter coupling between scene objects and gameplay logic. This category is well-suited for evaluating robustness under complex interaction flows, including dependency correctness and the reliability of generated code.

4.1.2 Settings

All experiments are conducted in Unreal Engine 5, which is one of the most advanced and broadly adopted game engines. For LLMs, we use Qwen-Plus as the default backbone for generation and evaluation. In Q2, we instead use GPT-4o only for evaluation to align with baslines.

4.2 Overall Game Evaluation (Q1)

To objectively evaluate the overall quality of UE games generated end-to-end by AutoUE, we adopt an LLMs-as-a-judge evaluation method and carefully design the evaluation prompt (the prompt de-

Approach	GAS
Generative Baselines	
DreamFusion (ICLR 23) (Poole et al., 2023)	4.83
Magic3D (CVPR 23) (Lin et al., 2023)	6.39
WonderJ (CVPR 24) (Yu et al., 2024)	7.38
Procedural Baselines	
Infinigen (CVPR 23) (Raistrick et al., 2023)	6.61
3D-GPT (3DV 25) (Sun et al., 2025)	6.76
SceneX (AAAI 25) (Zhou et al., 2025)	7.31
UnrealLLM (ACL 25) (SongTang et al., 2025)	7.71
AutoUE	**7.8**

Table 2: The scene generation evaluation results.

tails are provided in Appendix Figure 16).

The LLMs score each game along three dimensions, each on a 1–10 scale: (1) **Scene**: evaluates the correctness of PCG based on build logs and PCG node chains, (2) **Gameplay**: evaluates whether gameplay and interactive logic work as intended, based on gameplay/interaction attributes and code, (3) **Visual**: evaluates the visually coherence of game with the description, based on scene and interaction screenshots. The three dimension scores are aggregated into a final **Game** score using weights of 0.35, 0.35, and 0.3, respectively.

Notably, **Scene** and **Gameplay** place strong emphasis on successful build as these two dimensions are used to assess whether AutoUE can generate complete, playable games end-to-end. In contrast, **Visual** is used to evaluate the quality of games.

As shown shown in Table 1, AutoUE achieves strong performance, indicating that the system can stably generate playable UE games end-to-end.

On the **Scene** dimension, Hard games score higher than Medium games because Hard tasks involve more complex PCG graphs, and successfully building these graphs yields higher scores. This also indicates that AutoUE can reliably handle more complex scene generation requirements.

On the **Visual** dimension, Easy games lower than Medium games due to content constrains limit visual richness. Hard games receive the lowest scores primarily because deviations in interactions from the intended behavior degrade visual quality. This demonstrates that the **Visual** metric is sensitive to the quality of the generated games.

Overall, the **Game** and **Gameplay** scores follow the expected difficulty trend, suggesting that the primary challenge at higher difficulty stems from the complexity of gameplay and interaction logic.

4.3 Scene Generation Evaluation (Q2)

To further evaluate the quality of scenes generated by AutoUE, we follow the evaluation setup of UnrealLLM (SongTang et al., 2025). Specifically, UnrealLLM uses the GPT Aesthetic Score (GAS) to quantify aesthetic quality by assessing factors such as composition, color harmony, lighting, material and texture fidelity, richness of details, overall coherence, and artistic impact.

Variant	Category	S_{Node}	S_{Param}	S_{Pin}	S_{PCG}
W/o patterns	Easy	71.7	99.6	51.4	4.02
W/o params		100	47.9	100	3.96
W/o conns		100	100	82.1	5.34
AutoUE		100	100	100	**8.38**
W/o patterns	Medium	82.5	100	70.7	4.71
W/o params		100	53.1	99.7	4.49
W/o conns		100	100	79.4	5.09
AutoUE		100	100	100	**8.46**
W/o patterns	Hard	82	99.1	65.9	5.29
W/o params		100	61.5	100	5.18
W/o conns		100	100	80.2	4.75
AutoUE		100	100	100	**8.40**
W/o patterns	All	79.6	99.6	64.0	4.77
W/o params		100	55.2	99.9	4.63
W/o conns		100	100	80.4	5.02
AutoUE		100	100	100	**8.42**

Table 3: PCG graph evaluation results.

As shown in Table 2, AutoUE achieves the best GAS of 7.8, outperforming both generative and procedural baselines, which indicates that AutoUE produces scenes with higher-quality aesthetics.

Compared with generative baselines, procedural methods exhibit more stable scores, suggesting that procedural tools provide stronger structural priors and controllability, thereby enabling more consistent generation of high-quality scenes. However, Infinigen, 3D-GPT, and SceneX synthesize scenes in modeling environments such as Blender, rather than fully leveraging the PCG tool chain within game engines. In contrast, UnrealLLM grounds procedural generation directly in UE PCG graphs and achieves a high GAS of 7.71, which highlights the advantage of engine-native PCG pipelines in organizing scene layout.

Although AutoUE shows only a limited improvement over UnrealLLM, it is worth noting that GAS is a subjective metric and exhibits a pronounced saturation effect. In our empirical evaluation, we find it very difficult to obtain ratings above 8, which means that scores close to 8 typically indicate that the artistic quality has reached a high level. Furthermore, as illustrated in Figure 3, AutoUE produces scenes with richer object composition and more actionable interactive affordances than SceneX, leading to more coherent gameplay-oriented layouts.

4.4 PCG Graph Evaluation (Q3)

To verify the effectiveness of our defined PCG patterns and the RAG-based node specification, we conduct an ablation study on PlayGen-20. Specifically, we design three variants and four metrics:

- **W/o patterns** removes two pre-defined PCG node chains (Large objects and Small scatter objects) and freely constructs the PCG graph.
- **W/o params** removes all parameter content of PCG nodes from the PCG document corpus.
- **W/o conns** removes all pin connection content of PCG nodes from the PCG document corpus.

- S_{Node}: the node creation success rate, whether the nodes are defined PCG nodes.
- S_{Param}: the node's parameter filling success rate.
- S_{Pin}: the node's pin connection success rate.
- S_{PCG}: the LLMs-judged score that measures the quality of the PCG graph. It ranges from 1 to 10 and the prompt is provided in Appendix Fig 17).

As shown in Table 3, AutoUE achieves 100% success rates for node creation, parameter filling, pin connection across all difficulties, and obtains the highest S_{PCG} of 8.42, indicating robust and high-quality PCG graph generation. Meanwhile, each variant causes a clear degradation in its corresponding metric. **W/o patterns** substantially reduces S_{Node} and S_{Pin} to 79.6% and 64.0%, suggesting that the pre-defined patterns provide crucial structural priors for constructing PCG graphs. **W/o params** keeps the graph structure usable, but S_{Param} drops sharply to 55.2%, accompanied by a significant decrease in S_{PCG}, demonstrating that retrieved parameter semantics are essential for correct procedural generation behavior. **W/o conns** mainly affects wiring correctness: S_{Pin} decreases to 80.4%, and S_{PCG} drops to 5.02, confirming that RAG-provided pin-connection knowledge improves the reliability of PCG graph connections.

Collectively, these experimental results demonstrate the importance of our defined PCG patterns and the RAG-based node specification for reliably constructing PCG graphs in UE.

4.5 Gameplay Code Evaluation (Q4)

To verify the effectiveness of the game development designs in gameplay code generation, we also design three variants and four evaluation metrics:

- **W/o dependencies** removes module-dependency knowledge in $\mathcal{P}lan_{Code}$ and $\mathcal{G}enerate_{Code}$.
- **W/o code templates** removes the code templates in the prompt of $\mathcal{G}enerate_{Code}$.
- **W/o engine constraints** removes the engine constraints in the prompt of $\mathcal{G}enerate_{Code}$.

Variant	Category	MAS	MCS	IIS	GCS
W/o dependencies	Easy	8.20	0	0	2.46
W/o code templates		8.00	0	0	2.40
W/o engine constraints		8.00	0	0	2.40
AutoUE		8.00	7.50	6.70	**7.33**
W/o dependencies	Medium	8.21	0	0	2.46
W/o code templates		8.36	0	0	2.51
W/o engine constraints		8.14	0	0	2.44
AutoUE		8.07	7.57	6.79	**7.41**
W/o dependencies	Hard	8.38	0	0	2.51
W/o code templates		8.19	0	0	2.46
W/o engine constraints		8.13	0	0	2.44
AutoUE		8.06	7.56	6.73	**7.38**
W/o dependencies	All	8.28	0	0	2.48
W/o code templates		8.20	0	0	2.46
W/o engine constraints		8.10	0	0	2.44
AutoUE		8.05	7.55	6.74	**7.38**

Table 4: The gameplay code evaluation results.

- **MAS** (Module Analysis Score): evaluates the quality of the planned module attributes list.
- **MCS** (Module Code Score): evaluates the quality of the generated gameplay module code.
- **IIS** (Interactive Integration Score): evaluates how effectively interactions integrate with the module.
- **GCS** (Gameplay Code Score): evaluates the overall gameplay code generation quality, computed as $0.3 \times$ **MAS** $+ 0.3 \times$ **MCS** $+ 0.4 \times$ **IIS**.

These metrics are evaluated by LLMs judges using the prompt in Appendix Figure 18). **MAS**, **MCS**, and **IIS** range from 1–10 when compilation succeeds; otherwise **MCS** and **IIS** are set to 0.

As shown in Table 4, all three variants consistently fail to compile, which leads to a significant drop in **GCS**. This indicates that removing module dependencies, code templates, or engine constraints severely undermines the ability to generate compilable UE gameplay code. Overall, these results verify that our game development designs in gameplay code generation are essential for end-to-end compilable gameplay code generation.

5 Conclusion

In this work, we present a novel multi-agent system, AutoUE, for the automated generation of complete 3D games in UE. By leveraging RAG, game design patterns, and engine-specific constraints, AutoUE effectively coordinates agents to generate coherent scenes, robust gameplay code, and interactive objects, while ensuring the dynamic evaluation of gameplay through automated play-testing. Our experimental results demonstrate that AutoUE produces high-quality 3D games. This work also provides an extensible infrastructure for future advancements in automated game development.

Ethical Analysis

There are no ethical concerns associated with this work. AutoUE is a tool designed to automate 3D game generation in UE, with the aim of helping developers accelerate the game development process. The dataset used for model retrieval contains only publicly available 3D models, and the game generation tasks are based on predefined descriptions, without involving any sensitive or personal data.

Limitations

One limitation of our work is the inability to compare AutoUE with DreamGarden (Earle et al., 2025), an advanced 3D game generation system based on UE, as it has not been open-sourced. Additionally, although our system performs well across various tasks, the scope of our evaluation dataset may not fully encompass the complexity of real-world game development scenarios. Therefore, further evaluation on a broader set of tasks and more challenging game scenarios would help assess the scalability and robustness of AutoUE.

References

Jake Bruce, Michael D Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, and 1 others. 2024. Genie: Generative interactive environments. In *Forty-first International Conference on Machine Learning*.

Steph Buongiorno, Lawrence Klinkert, Zixin Zhuang, Tanishq Chawla, and Corey Clark. 2024. Pangea: procedural artificial narrative using generative ai for turn-based, role-playing video games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, pages 156–166.

Haoxuan Che, Xuanhua He, Quande Liu, Cheng Jin, and Hao Chen. 2025. Gamegen-x: Interactive open-world game video generation. In *The Thirteenth International Conference on Learning Representations*.

Peng Chen, Pi Bu, Jun Song, Yuan Gao, and Bo Zheng. 2024. Can vlms play action role-playing games? take black myth wukong as a study case. In *NeurIPS 2024 Workshop on Open-World Agents*.

Marc-Alexandre Côté, Akos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, and 1 others. 2018. Textworld: A learning environment for text-based games. In *Workshop on Computer Games*, pages 41–75. Springer.

Shiqi Dai, Xuanyu Zhu, Naiqi Li, Tao Dai, and Zhi Wang. 2024. Procedural level generation with diffusion models from a single example. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 10021–10029.

Sam Earle, Samyak Parajuli, and Andrzej Banburski-Fahey. 2025. Dreamgarden: A designer assistant for growing games from a single prompt. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, pages 1–19.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Amna Hassan. 2025. Automated unity game template generation from gdds via nlp and multi-modal llms. *arXiv preprint arXiv:2509.08847*.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7903–7910.

Shun-ichiro Hayashi, Daichi Mukunoki, Tetsuya Hoshino, Satoshi Ohshima, and Takahiro Katagiri. 2025. 3dify: a framework for procedural 3d-cg generation assisted by llms using mcp and rag. *arXiv preprint arXiv:2510.04536*.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851.

Chengpeng Hu, Yunlong Zhao, and Jialin Liu. 2024. Game generation via large language models. In *2024 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.

Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. 2023. Magic3d: High-resolution text-to-3d content creation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 300–309.

Mahdi Farrokhi Maleki and Richard Zhao. 2024. Procedural content generation in games: A survey with insights on emerging llm integration. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, pages 167–178.

Luiza Martins de Freitas Cintra, Elisa Ayumi Masasi de Oliveira, Rafael Teixeira Sousa, Valdemar Vicente Graciano, Arlindo Galvão, Gustavo Webster, and Sofia Larissa da Costa Paiva. 2025. An llm-enhanced framework for bridging simulators and game engines towards realistic 3d simulations. In *Proceedings of the 2025 ACM International Conference on Interactive Media Experiences*, pages 402–405.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Ben Poole, Ajay Jain, Jonathan T Barron, and Ben Mildenhall. 2023. Dreamfusion: Text-to-3d using 2d diffusion. In *The Eleventh International Conference on Learning Representations*.

Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, and 1 others. 2023. Infinite photorealistic worlds using procedural generation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12630–12641.

SongTang SongTang, Kaiyong Zhao, Lei Wang, Yuliang Li, Xuebo Liu, Junyi Zou, Qiang Wang, and Xiaowen Chu. 2025. Unrealllm: Towards highly controllable and interactable 3d scene generation by llm-powered procedural content generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19417–19435.

Shyam Sudhakaran, Miguel González-Duque, Matthias Freiberger, Claire Glanois, Elias Najarro, and Sebastian Risi. 2023. Mariogpt: Open-ended text2level generation through large language models. *Advances in Neural Information Processing Systems*, 36:54213–54227.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 2025. 3d-gpt: Procedural 3d modeling with large language models. In *2025 International Conference on 3D Vision (3DV)*, pages 1253–1263. IEEE.

Yuqian Sun, Zhouyi Li, Ke Fang, Chang Hee Lee, and Ali Asadipour. 2023. Language as reality: a co-creative storytelling game experience in 1001 nights using generative ai. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pages 425–434.

Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, and 1 others. 2025. Lumine: An open recipe for building generalist agents in 3d open worlds. *arXiv preprint arXiv:2511.08892*.

Graham Todd, Alexander G Padula, Matthew Stephenson, Éric Piette, Dennis J Soemers, and Julian Togelius. 2024. Gavel: Generating games via evolution and language models. *Advances in Neural Information Processing Systems*, 37:110723–110745.

Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. 2025. Diffusion models are real-time game engines. In *The Thirteenth International Conference on Learning Representations*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*.

Shu Wang, Lei Ji, Renxi Wang, Wenxiao Zhao, Haokun Liu, Yifan Hou, and Ying Nian Wu. 2025. Explore the reasoning capability of llms in the chess testbed. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 611–622.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Shawn Ma, and Yitao Liang. 2023. Describe, explain, plan and select: interactive planning with llms enables open-world multi-task agents. *Advances in Neural Information Processing Systems*, 36:34153–34189.

Kazuki Wayama, Tomoyuki Yokogawa, Sousuke Amasaki, Hirohisa Aman, and Kazutami Arimoto. 2022. Verifying game logic in unreal engine 5 blueprint visual scripting system using model checking. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–8.

Yongzhi Xu, Yonhon Ng, Yifu Wang, Inkyu Sa, Yunfei Duan, Yang Li, Pan Ji, and Hongdong Li. 2024. Sketch2scene: Automatic generation of interactive 3d game scenes from user's casual sketches. *arXiv preprint arXiv:2408.04567*.

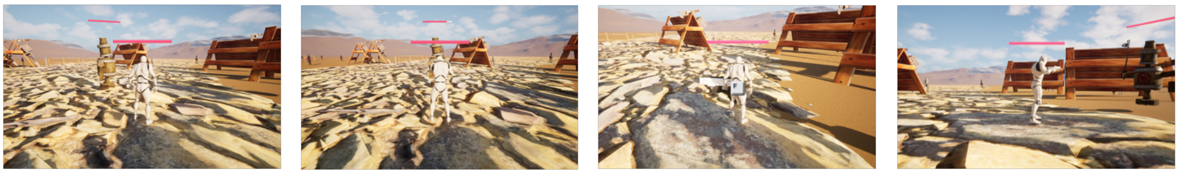
Runxin Yang, Yuxuan Wan, Shuqing Li, and Michael R Lyu. 2025. 90% faster, 100% code-free: Mllm-driven zero-code 3d game development. *arXiv preprint arXiv:2509.26161*.

Hong-Xing Yu, Haoyi Duan, Junhwa Hur, Kyle Sargent, Michael Rubinstein, William T Freeman, Forrester Cole, Deqing Sun, Noah Snavely, Jiajun Wu, and 1 others. 2024. Wonderjourney: Going from anywhere to everywhere. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6658–6667.

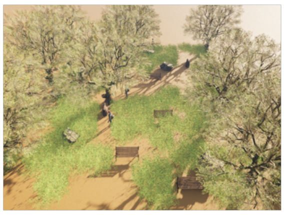
Yibo Zhang, Li Zhang, Rui Ma, and Nan Cao. 2025. Texverse: A universe of 3d objects with high-resolution textures. *arXiv preprint arXiv:2508.10868*.

Mengqi Zhou, Yuxi Wang, Jun Hou, Shougao Zhang, Yiwei Li, Chuanchen Luo, Junran Peng, and Zhaoxiang Zhang. 2025. Scenex: Procedural controllable large-scale scene generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 10806–10814.

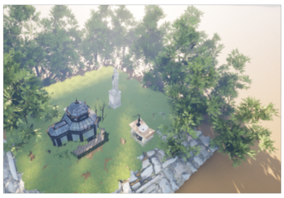
A Appendix



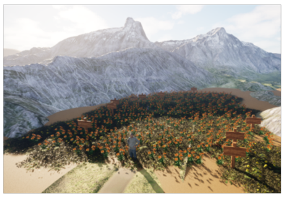
The game takes place in a small outdoor training yard. The area is enclosed by wooden fences. Interactive objects include: TrainingDummy: Used for basic interaction. WeaponRack: Holds a simple weapon. InstructorNPC: Standing near the fence. The ground and fences are non-interactive decorations. The player's objective is to complete basic training actions. The core gameplay includes: Movement The player can move freely in the yard. Use Training Dummy (TrainingDummy) Interacting with the dummy triggers a simple response. Take Weapon (WeaponRack) The player can pick up a weapon.



The game takes place at the edge of a forest in an open rest spot during the daytime. Sunlight filters through the trees, creating a calm and safe atmosphere. Interactive objects include: RestBench: Placed near the trees and can be used to recover health. Backpack: Lying on the ground and can be picked up. ScoutNPC: Standing near the forest entrance. Trees, bushes, and rocks form the natural background of the scene. The player's objective is to recover and prepare before entering the forest. Core gameplay includes: Movement The player can move freely around the rest spot. Rest and Heal (RestBench) Interacting with the bench allows the player to rest for a short time, gradually restoring health. This provides a safe opportunity to recover before facing potential dangers in the forest. Pick Up Backpack (Backpack) The player can collect the backpack to prepare for the upcoming journey.



The game takes place in a small garden courtyard. Flowers and paths fill the area. Interactive objects include: WaterPump: Can be used to water plants. FlowerBed: Can be examined. GardenerNPC: Standing near the path. Trees and statues act as non-interactive decorations. The player's objective is to interact with the garden. The core gameplay includes: Movement The player can walk around the courtyard. Use Water Pump (WaterPump) Interacting with the pump triggers a simple action. Examine Flowers (FlowerBed) The player can inspect the flower bed.



The game takes place in an open grassy field during the daytime. The area is wide and flat, with clear visibility. Interactive objects include: SupplyCrate: Placed near the center of the field and can be opened. SignPost: Standing beside a dirt path and can be read. FieldNPC: Standing near the signpost and available for interaction. Grass, flowers, and distant hills act as non-interactive decorations. The player's objective is to explore the field and perform basic interactions. The core gameplay includes: Movement The player can move freely across the field. Open Supply Crate (SupplyCrate) The player can open the crate to obtain items. Read Sign (SignPost) Interacting with the sign displays simple information.

The game takes place in a standard classroom during the daytime. Rows of desks face a whiteboard at the front of the room. Interactive objects include: Whiteboard: Located at the front wall and can be inspected. TeacherDesk: Placed near the whiteboard and can be opened. Chair: Positioned beside a desk and can be interacted with. Windows, posters, and ceiling lights act as non-interactive decorations. The player's objective is to explore the classroom environment. The core gameplay includes: Movement The player can move freely around the classroom. Inspect Whiteboard (Whiteboard) Interacting displays simple written information. Open Teacher Desk (TeacherDesk) The player can check items inside.

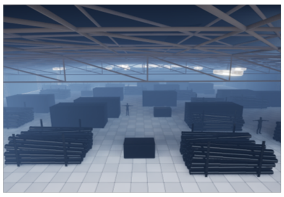
Figure 4: The descriptions and screenshots of all **Easy** games.

The game takes place in a small snowy campsite. A tent stands at the center of the scene. Interactive objects include: Tent: Can be entered or exited. Campfire: Located in front of the tent and can be lit. SupplyBag: Placed beside the tent and can be checked. Snow-covered trees and rocks act as decorations. The player's objective is to secure the campsite. The core gameplay includes: Movement The player can move around the campsite. Light Campfire (Campfire) Interacting with the campfire provides warmth. Check Supply Bag (SupplyBag) The player can inspect the bag for items.

The game takes place in a small abandoned farmyard. At the center of the scene stands an old barn, with a few interactive objects placed nearby: Crate: Located near the barn entrance. It cean be opened to obtain supplies. Lantern: Hanging beside the barn door. It can be turned on or off. FarmGuard: Patrols slowly around the farmyard and reacts when the player gets close. The surrounding area contains fences, hay piles, and grass as non-interactive decorations. The player character can move freely and interact with objects when nearby. The player's objective is to explore the farmyard and perform basic interactions. The core gameplay includes: Movement The player can move freely around the farmyard. Open Crate (Crate) When approaching the crate, the player can open it to collect items. Use Lantern (Lantern) Interacting with the lantern changes the lighting state of the area.

The game takes place on a small wooden dock by the sea. A moored boat sits beside the dock. Interactive objects include: Boat: Docked at the pier and can be boarded. FishingCrate: Placed near the edge of the dock and can be opened. DockWorkerNPC: Standing near the boat for interaction. The sea, ropes, and wooden posts serve as non-interactive decorations. The player's objective is to prepare for departure. The core gameplay includes: Movement The player can walk along the dock. Open Fishing Crate (FishingCrate) The player can collect items from the crate. Board Boat (Boat) Interacting with the boat triggers a transition event.

The game takes place at a small desert outpost surrounded by sand dunes. A simple shelter stands at the center. Interactive objects include: WaterBarrel: Placed beside the shelter and can be used to refill water. SupplyChest: Located inside the shelter and can be opened. OutpostGuard: Standing near the entrance and observing the area. The background contains dunes and rocks as decorative elements. The player's objective is to resupply at the outpost. The core gameplay includes: Movement The player can move freely around the outpost. Use Water Barrel (WaterBarrel) Interacting with the barrel restores resources. Open Supply Chest (SupplyChest) The player can open the chest to obtain supplies.



The game takes place in a small underground storage room. The space is narrow and dimly lit. Interactive objects include: LightSwitch: Mounted on the wall and controls the room lighting. StorageBox: Placed on the floor and can be opened. MaintenanceNPC: Standing near the exit. Pipes and concrete walls form the non-interactive environment. The player's objective is to inspect the storage room. The core gameplay includes: Movement The player can move within the room. Toggle Light (LightSwitch) Interacting with the switch changes lighting. Open Storage Box (StorageBox) The player can open the box to inspect its contents.

The game takes place at a small camp on flat ground. A tent is set up at the center of the area. Interactive objects include: Tent: Can be entered or exited. Campfire: Can be lit or extinguished. SupplyBox: Placed beside the tent and can be opened. The background includes grass and distant mountains. The player's objective is to prepare the camp. The core gameplay includes: Movement The player can move around the camp. Light Campfire (Campfire) Interacting with the campfire changes its state. Open Supply Box (SupplyBox) The player can collect items from the box.

The game takes place in a small clearing beside a lake during the daytime. The water is calm and reflective. Interactive objects include: WoodenDock: Extends slightly into the lake and can be used. StorageBox: Placed near the shore and can be opened. CamperNPC: Standing near the lake. Water, grass, and trees decorate the area. The player's objective is to explore the lakeside. The core gameplay includes: Movement The player can walk around the clearing. Use Dock (WoodenDock) Interacting with the dock triggers a simple action. Open Storage Box (StorageBox) The player can inspect the box contents.

Figure 5: The descriptions and screenshots of all **Medium** games.

Figure 6: The descriptions and screenshots of all **Hard** games.

The Prompt Details of $\mathcal{Decompose}$

Role: You are a game design and procedural content analysis expert.
Task: Decompose the user's natural language input into two clear parts:
1. Scene Description: Describe environment, theme, terrain, climate, time of day, atmosphere, visual style, etc.
2. Gameplay Description: Describe what the player can do, interaction logic, system modules, gameplay mechanics, and objectives.
Input Example: I want a forest campsite scene where the player can collect wood, light a fire, and build tents.
Output Format:
```
{
    scene_description: A forest campsite with a natural environment, a calm atmosphere, and elements such as trees,
                       tents, and a campfire.,
    gameplay_description: The player can collect wood, light fires, and build tents through interactive actions.
}
```

Figure 7: The prompt details of $\mathcal{Decompose}$ in Section 3.2.

The Prompt Details of the first step in $\mathcal{Generate}_{Obj}$

Role: You are an expert in scene description generation.
Task: Based on the user's natural language description, generate a formal scene description JSON.
Output Format: Please strictly follow the structure below (output must be valid JSON):
```
{
    "scene_name": "A short scene title",
    "theme": "Theme (e.g., ancient castle, sci-fi city, forest campsite, etc.)",
    "style": "Art style (e.g., realistic, cartoon, low-poly, etc.)",
    "environment": {
        "terrain_type": "Terrain type (mountain, plain, desert, coast, etc.)",
        "climate": "Climate (sunny, rainy, night, foggy, etc.)",
        "lighting": "Lighting type (morning light, sunset, night lighting, dynamic, etc.)"
    },
    "main_elements": ["Primary elements such as trees, buildings, roads, water bodies, rocks, etc."],
    "mood": "Overall atmosphere description (peaceful, mysterious, lively, oppressive, etc.)"
}
```

The Prompt Details of the second step in $\mathcal{Generate}_{Obj}$

Role: You are a level analyst and a procedural generation system designer.
Task: Analyze the formal scene description provided by the upstream node, identify the key model types that should appear in the scene, and recommend a reasonable generation method (placement_method) and placement description for each model.
At the same time, for every model you must generate an English field `prompt_for_embedding`, which will be used to retrieve similar models from a 3D model embedding vector database.
In addition, each model must include a `category` field, whose value must be selected from the following fixed list:
```
[
    'Art & Abstract', 'Fashion & Style', 'Characters & Creatures', 'Architecture', 'Science & Technology', 'Animals & Pets',
    'Nature & Plants', 'Places & Travel', 'Cultural Heritage & History', 'Furniture & Home', 'Weapons & Military', 'People',
    'Electronics & Gadgets', 'Food & Drink', 'Cars & Vehicles', 'Music', 'Sports & Fitness', 'News & Politics'
]
```
Please choose the most appropriate category based on the semantic meaning and intended usage of the model.
Input Example:
```
{
    "scene_name": "Forest Camp", "theme": "Nature Exploration", "style": "Realistic",
    "environment": {"terrain_type": "Forest", "climate": "Clear", "lighting": "Morning Light"},
    "main_elements": ["Trees", "Grass", "Tent", "Campfire", "Cabin", "Road", "Water Body"],
    "mood": "Peaceful and warm"
}
```
Output Format:
```
{
  "models": [
    {
      "name": "OakTree",
      "category": "Nature & Plants",
      "placement_method": "surface_sampling",
      "placement_description": "Distributed across forest terrain at medium density, aligned to terrain normals.",
      "prompt_for_embedding": "Oak tree, forest terrain, morning light, realistic bark, dense leaves"
    }
  ]
}
```

Figure 8: The prompt details of $\mathcal{Generate}_{Obj}$ in Section 3.3.

The Prompt Details of $\mathcal{P}lan_{Node}$

Role: You are an expert in Unreal Engine 5 Procedural Content Generation (PCG) systems.
Task: Your responsibility is to generate a complete PCG graph structure (JSON) based on the input model information and node definitions.

- The system will provide:
- Parameter Space descriptions for each node
- Pin definitions (Input Pins / Output Pins)
- A list of available interactive actor class names that can be used by PCGSpawnActorSettings nodes:

InteractiveObjectClassList: {{extra_input}}

Output Format :

1. Output format (STRICT JSON, no code blocks, no explanatory text):

```
{
  "model": "<from input model field>",
  "pattern_desc": "<from input pattern_desc field>",
  "nodes": [
    { "type": "PCGGetLandscapeSettings@[0,0]", "parameters": { ... } },
    ...
  ],
  "connections": [
    "0.Out|1.In"
  ]
}
```

2. For PCGSpawnActorSettings:
- The parameters MUST include a field named "TemplateActor"
- TemplateActor MUST be selected from InteractiveObjectClassList
- Do NOT invent or fabricate class names

3. PCGTransformPointsSettings special rule (MANDATORY):
- When the node type is `PCGTransformPointsSettings`, parameters MUST include:
  - RotationMin (FRotator)
  - RotationMax (FRotator)
- Rotation is ONLY allowed around the Z axis (Yaw):
  - Pitch MUST be 0
  - Roll MUST be 0
- The format of RotationMin / RotationMax MUST be: {"X": 0, "Y": 0, "Z": <value>}

Rotation constraints (ENFORCED):
- Only random rotation around Z axis is allowed
- X (Pitch) MUST always be 0
- Y (Roll) MUST always be 0
- No non-zero randomness is allowed on X / Y axes

4. Connection generation rules (CRITICAL):
- You MUST generate connections strictly based on node Pins (Input Pins / Output Pins). Do NOT guess connections.
- Connection string format MUST be: "A.<OutputPinName>|B.<InputPinName>"
- Connection principles:
  1. For each output pin, find the FIRST compatible input pin in subsequent nodes
  2. Types must be strictly or loosely compatible (Point → Point, Spatial → Spatial)
  3. Prefer pins that appear earlier in the pin list
  4. If no compatible input pin exists, do NOT create a connection
  5. Cross-model connections are NOT allowed
- Connection order MUST be deterministic (increasing output node index)

5. Parameter filling rules:
- parameters may ONLY use fields that exist in the parameter descriptions
- Prefer default values or explicitly listed optional values
- If uncertain, use conservative example values (e.g. 1, true, false)
- Do NOT introduce nonexistent structures or field names

6. Pin parsing requirements:
- Pin content may be plain text or structured JSON
- You MUST parse Input Pin / Output Pin names and types
- If pin information is insufficient, generate only the most obvious Out → In connection

Example:

PCGSurfaceSamplerSettings.Out (Point) → PCGPointFilterSettings.In (Point)
Connection string:
"0.Out|1.In"

Final Reminder:
- ALL rules must be satisfied simultaneously
- Z-axis random rotation for TransformPoints is MANDATORY
- Conservative, deterministic, and reproducible behavior is preferred

Now wait for the user-provided model JSON and generate the final valid JSON.

Figure 9: The prompt details of $\mathcal{P}lan_{Node}$ in Section 3.4.1.

The Prompt Details of $\mathcal{G}enerate_{Node}$

Role: You are an Unreal Engine 5 Procedural Content Generation (PCG) Graph composition expert.
Task: Based on the Object Generation Plan, generate one PCG node chain per model, strictly following the predefined PCG node pattern library.
The final output will be used to automatically generate PCG Graphs in Unreal Engine,
therefore it must meet production-level stability requirements.
CRITICAL ONE-TO-ONE MAPPING RULE (VERY IMPORTANT):
1. The "Object Generation Plan" contains a list called "models".
2. For EACH model entry in that list, you MUST generate EXACTLY ONE
corresponding item in the output JSON array.
3. The output array length MUST be exactly equal to the number of input models.
4. The value of the output field "model" MUST EXACTLY MATCH the input
models[i].name string (case-sensitive, no renaming).
5. You are STRICTLY FORBIDDEN to:
   - Skip any model
   - Merge multiple models into one output item
   - Generate extra output items that do not correspond to an input model

This is a STRICT one-to-one relationship.
STRICT RULES — Node connections MUST strictly follow the rules below:
You MUST choose ONE of the following two patterns.
You are NOT allowed to create new patterns, mix patterns, or change node order.
1. Large Actor Placement
The node chain MUST be EXACTLY:
PCGGetLandscapeSettings → PCGSurfaceSamplerSettings → PCGTransformPointsSettings →
PCGSelectRandomPointsSettings → PCGBoundsModifierSettings → PCGSelfPruningSettings → PCGSpawnActorSettings
2. Small Actor Placement with Exclusion
The node chain MUST be EXACTLY:
PCGGetLandscapeSettings → PCGSurfaceSamplerSettings → PCGTransformPointsSettings →
PCGSelectRandomPointsSettings → PCGBoundsModifierSettings → PCGSelfPruningSettings →
PCGDifferenceSettings → PCGCollapseSettings → PCGSpawnActorSettings
NOT allowed:
- Adding new nodes
- Removing any node
- Changing node order
- Creating a third pattern
- Merging the two patterns
- Using any node not listed above

Input Example:
You will receive the following two parts as input:
1. Scene Description
{scene_description}
2. Object Generation Plan (JSON)
Example:
```
{
  "models": [
    {
      "name": "OakTree",
      "placement_method": "surface_sampling",
      "placement_description": "Distributed at medium density on forest terrain."
    },
    {
      "name": "RockSmall",
      "placement_method": "surface_sampling_with_exclusion",
      "placement_description": "Scattered sparsely, avoiding trees."
    }
  ]
}
```
Output Format:
The output MUST be a strict JSON array.
Do NOT include comments, explanations, or any extra natural language.
For an input with N models, the output array MUST contain EXACTLY N items.
Each item MUST correspond to exactly ONE input model.
Correct One-to-One Output Example:
Input models:
["OakTree", "RockSmall"]
Correct output:
```
[
 {
  "model": "OakTree",
  "pattern_desc": "Large Actor Placement@Medium density trees aligned to terrain",
  "nodes": [
   "PCGGetLandscapeSettings@[0,0]",
   "PCGSurfaceSamplerSettings@[200,0]",
   "PCGTransformPointsSettings@[400,0]",
   "PCGSelectRandomPointsSettings@[600,0]",
   "PCGBoundsModifierSettings@[800,0]",
   "PCGSelfPruningSettings@[1000,0]",
   "PCGSpawnActorSettings@[1200,0]"
  ]
 },
 {
  "model": "RockSmall",
  "pattern_desc": "Small Actor Placement with Exclusion@Sparse rocks avoiding trees",
  "nodes": [
   "PCGGetLandscapeSettings@[0,200]",
   "PCGSurfaceSamplerSettings@[200,200]",
   "PCGTransformPointsSettings@[400,200]",
   "PCGSelectRandomPointsSettings@[600,200]",
   "PCGBoundsModifierSettings@[800,200]",
   "PCGSelfPruningSettings@[1000,200]",
   "PCGDifferenceSettings@[1200,200]",
   "PCGCollapseSettings@[1400,200]",
   "PCGSpawnActorSettings@[1600,200]"
  ]
 }
]
```

Figure 10: The prompt details of $\mathcal{G}enerate_{Node}$ in Section 3.4.2.

The Prompt Details of $\mathcal{P}lan_{Code}$

Role: You are a senior Unreal Engine 5 game architecture designer.
Task: Based on the gameplay feature description provided by the user, analyze and design the concrete gameplay logic modules (Modules) required to fully and correctly implement ALL described gameplay behavior.
These modules will be invoked inside the HandleInteraction() function of interactive objects (AInteractiveObjectBase) to execute concrete gameplay logic (such as item pickup, healing, trading, quest triggering, environment interaction, etc.).
Existing Modules — MUST NOT Be Generated:
The current project already includes:
- BattleModule (combat logic)
- InteractionModule (interaction framework logic)

You must NOT generate, mention, or overlap with these modules in any form.
CRITICAL MODULE COMPLETENESS RULES:
1. You MUST generate AT LEAST 3 core gameplay modules.
2. Every generated module MUST directly serve the provided gameplay_description and contribute to implementing it.
3. The set of generated modules, when combined, MUST be sufficient to fully realize ALL gameplay behaviors described by the user. No gameplay requirement may be left unimplemented.
4. Do NOT generate placeholder, generic, or filler modules. Each module must have a clear, concrete responsibility.
5. If the gameplay_description is simple, you MUST still decompose it into multiple meaningful gameplay modules (minimum 3), by separating responsibilities where appropriate (e.g., state management, reward handling, environment effects).

CRITICAL DEPENDENCY & ORDERING RULES (VERY IMPORTANT):
The output JSON field "core_modules" will be used DIRECTLY to generate C++ module code in the given order.
Therefore:
1. The order of modules in "core_modules" MUST strictly follow dependency-safe generation order (topological order).
2. If Module B depends on Module A:
   - Module A MUST appear BEFORE Module B
   - Module B MUST explicitly list "ModuleA" in its "dependencies" array
3. A module is ONLY allowed to depend on modules that appear earlier.
4. Forward dependencies, circular dependencies, or implicit dependencies are STRICTLY FORBIDDEN.
5. If a module has no dependencies, its "dependencies" field MUST be [ ].

Strictly Forbidden Modules or Responsibilities
- InteractionModule / InteractionSystem / InteractionManager
- BattleModule / CombatModule / DamageModule
- Input handling, hit detection, combat resolution, AI logic

Assume interaction triggering and combat execution already exist.
What You SHOULD Generate
ONLY generate concrete gameplay logic modules, such as:
- InventoryModule
- QuestModule
- DialogueModule
- TradeModule
- HealModule
- EnvironmentModule
- RewardModule
- ConstructionModule

etc.
Output Format:
```
{
 "game_concept": "High-level summary of the gameplay concept",
 "core_modules": [
  {
   "module_name": "ModuleName",
   "purpose": "Clear and concrete responsibility",
   "interaction_usage": "How this module is invoked from HandleInteraction()",
   "key_classes": ["FModuleName"],
   "dependencies": []
  }
 ]
}
```
Requirements:
1. Generate AT LEAST 3 modules.
2. All modules must directly serve gameplay_description.
3. Dependencies must be explicit, minimal, and correct.
4. core_modules MUST already be dependency-sorted.
5. Output MUST be directly parsable JSON only.

Figure 11: The prompt details of $\mathcal{P}lan_{Code}$ in Section 3.5.

The Prompt Details of $\mathcal{G}enerate_{Code}$

Role: You are a senior Unreal Engine 5 C++ module development engineer.
Task: Your task is to generate the corresponding C++ implementation files (.h / .cpp) for each module based on the JSON output from the previous node (ModuleAnalyzer).
All generated module code MUST strictly follow the format below.
You must NOT add or remove blank lines, and must NOT change function order or indentation style.
IMPORTANT EXPORT RULE (CRITICAL):
All generated module classes are part of the LLMEditor plugin public API.
Therefore, EVERY module class MUST be exported using the `LLMEDITOR_API` macro.
Correct example:
class LLMEDITOR_API FInventoryModule : public IModuleBase
Missing the export macro is considered a critical error.
Code Template Example:

```
CustomModule.h:
#pragma once
#include "CoreMinimal.h"
#include "IModuleBase.h"
class LLMEDITOR_API FCustomModule : public IModuleBase
{
public:
  virtual void Initialize() override
  {
    UE_LOG(LogTemp, Log, TEXT("[CustomModule] Initialized"));
  }
  virtual void Shutdown() override
  {
    UE_LOG(LogTemp, Log, TEXT("[CustomModule] Shutdown"));
    Items.Empty();
  }
  static FName StaticModuleName() { return TEXT("CustomModule"); }
public:
  /** Add item interface: increment count and print current inventory */
  void AddItem(const FString& ItemName);
  /** Print current inventory contents */
  void PrintInventory() const;
private:
  /** Stored inventory items */
  TMap<FString, int32> Items;
};
CustomModule.cpp:
......
```

Register module:
REGISTER_MODULE(FCustomModule)
Important Naming / Include Rules:

```
1. The .cpp file MUST include ONLY the following headers
   and MUST NOT include any other module headers:
#include "<ModuleName>.h"
#include "ModuleRegistry.h"
#include "ModuleMacro.h"
#include "MyModuleManager.h"
#include "EvaluationLogModule.h"
#include "CustomModules.h"
```

2. You MUST NOT include:
- Any other module headers
- Any dependency module headers
- Any forward includes to other modules

3. All cross-module access MUST rely on:
   #include "CustomModules.h"
4. Module class names MUST use the `F` prefix (e.g. FInventoryModule).
   The `A` prefix is strictly forbidden.
5. All module classes MUST include `LLMEDITOR_API`.
6. Header and source filenames MUST match the module name:
   InventoryModule.h / InventoryModule.cpp
7. All #include directives MUST reference the exact filename.
   Example:
   #include "InventoryModule.h" ✅
   #include "AInventoryModule.h" ❌
8. Formatting, indentation, comments, and blank lines MUST match the example exactly. Only names and business logic may differ.
9. Output MUST be strict JSON only, with no extra text:

```
{
 "modules": [
  {
   "module_name": "InventoryModule",
   "header_code": "Full .h file content",
   "source_code": "Full .cpp file content"
  }
 ]
}
```

Figure 12: The prompt details of $\mathcal{G}enerate_{Code}$ in Section 3.5.

The Prompt Details of $\mathcal{P}lan_{Int}$

Role: You are an expert in Unreal Engine 5 interaction systems and AI design.
Input: You will receive two inputs:
1. Output from Model Retrieval Agent (a list of models);
2. Output from Gameplay Code Generator (a list of modules).

Task: Based on the model names, their semantic meaning, and the registered modules, infer the interaction logic between each model and the player, and determine:

- Whether a battle system is required;
- Whether the object is a movable AI enemy unit that uses a navigation system.

Mandatory One-to-One Mapping Rule (STRICT):

- You MUST generate exactly one interactive_objects entry for EACH model provided by KeyElementExtractor.
- The total number of interactive_objects entries MUST be exactly equal to the number of input models.
- The model_name field MUST exactly match the corresponding model "name" from KeyElementExtractor (character-by-character).
- Do NOT omit, merge, split, rename, infer, or add any models.
- The output order of interactive_objects MUST strictly follow the input order from KeyElementExtractor.
- Even if a model appears non-interactive, it MUST still be represented with a valid entry.

Core Analysis Rules (Must Be Followed):

1. Interactive Object Analysis
    - Infer possible player interaction behaviors, such as: open, pick up, ignite, talk, attack, defend, trigger mechanisms, etc.
    - The interaction flow must follow the UE interaction chain: AInteractiveCharacter::TriggerInteract() → AInteractiveObjectBase::HandleInteraction() → Corresponding module interface calls (e.g., FInventoryModule::AddItem()).

2. Battle System Determination (requires_battle_system)

- Must be set to true in the following cases:
    - Hostile units
    - Objects that can attack or be attacked
    - Defensive structures, combat mechanisms, destructible combat objects
- Otherwise, set to false.

3. Movable AI Enemy Unit Determination (is_movable_ai_unit)

- Can only be true if the object satisfies ALL of the following conditions:
    - Is a hostile unit;
    - Moves within the scene;
    - Uses a navigation system (NavMesh / AIController);
    - Actively approaches, chases, or patrols around the player.
- Examples: Enemy_Guard, Monster, PatrolBot → true
- The following objects must be false:
    - Chests, campfires, mechanisms
    - Static turrets
    - Destructible but non-moving objects
    - Non-hostile NPCs (merchants, quest NPCs)

Mandatory Consistency Rule:

- If is_movable_ai_unit = true → requires_battle_system must also be true
- The reverse is not mandatory.

Output Format:

```
{
 "interactive_objects": [
  {
   "model_name": "ModelName",
   "interaction_summary": "One-sentence summary of the interaction",
   "linked_modules": ["RelatedModuleNames"],
   "interaction_flow": "TriggerInteract() → HandleInteraction() → FxxxModule::Interface()",
   "requires_battle_system": true or false,
   "is_movable_ai_unit": true or false
  }
 ]
}
```

Figure 13: The prompt details of $\mathcal{P}lan_{Int}$ in Section 3.6.

The Prompt Details of $\mathcal{G}enerate_{Code}$

Role: You are an Unreal Engine 5 C++ programmer.
Input Example: Your ONLY input consists of two parts (already concatenated by the system):
1) Gameplay Code Generator output (JSON), format:

```
{
  "modules": [
    {
      "module_name": "InventoryModule",
      "header_code": "...",
      "source_code": "..."
    }
  ]
}
```

2) Interactive Object Planner output (JSON), format:

```
{
  "interactive_objects": [
    {
      "model_name": "Chest",
      "interaction_summary": "The player opens the chest and adds items to the inventory",
      "linked_modules": ["InventoryModule"],
      "interaction_flow": "TriggerInteract() → HandleInteraction() → FInventoryModule::AddItem()",
      "requires_battle_system": false,
      "is_movable_ai_unit": false
    }
  ]
}
```

CRITICAL MODULE API VALIDATION RULES (VERY IMPORTANT):
You MUST strictly follow these rules when calling module functions:
1. You are ONLY allowed to call functions that ACTUALLY EXIST in the provided ModuleCodeGenerator output.
2. A function is considered "existing" ONLY IF:
- It is declared in the module's header_code
- OR it is implemented as a public method in the source_code

3. You MUST NOT:
- Invent function names
- Guess function signatures
- Call functions mentioned in interaction_flow if they do NOT exist in module code

IMPORTANT RULES (MUST BE STRICTLY FOLLOWED):
1. You MUST generate one interactive object class for EVERY entry in InteractiveObjectAnalyzer.interactive_objects.
Even if a model has no interaction flow (linked_modules is empty or interaction_flow is "no interaction"), you MUST still generate the class, and HandleInteraction must at least call: Super::HandleInteraction(InteractingCharacter);
2. ALL generated interactive object classes MUST be exported from the LLMEditor module. Therefore, EVERY generated class declaration MUST include the `LLMEDITOR_API` macro.
Correct format (MUST use exactly this pattern):

```
UCLASS()
class LLMEDITOR_API AMyObjectInteractiveObject : public AInteractiveObjectBase
{
   GENERATED_BODY()
   ...
};
```

FORBIDDEN:
- Missing LLMEDITOR_API
- Using any other *_API macro
- Attaching the macro to the wrong identifier

3. File naming and includes MUST strictly follow these rules:
- Class name: `A<ModelName>InteractiveObject`
- Header file: `<ModelName>InteractiveObject.h` and MUST include: `#include "<ModelName>InteractiveObject.generated.h"`
- Source file: `<ModelName>InteractiveObject.cpp` and MUST include: `#include "<ModelName>InteractiveObject.h"` `#include "CustomModules.h"`

4. Subclass responsibilities and restrictions:
- Must inherit from `AInteractiveObjectBase`
- MUST NOT:
  - Inherit from `IBattleInterface` (handled by base class)
  - Create `BattleStats`
  - Bind death events (handled by base class)
- Subclass is ONLY responsible for:
  - Setting `bIsPartOfBattleSystem`
  - Overriding `HandleInteraction` and implementing interaction logic (calling modules)
  - Optionally overriding `BeginPlay` and calling `Super::BeginPlay()` when needed

5. Manager access and module calls MUST use this exact pattern:

```cpp
auto& Manager = FMyModuleManager::Instance();
auto* Inventory = Manager.GetModule<FInventoryModule>();
if (Inventory)
{
   Inventory->AddItem(GetName());
}
```

IMPORTANT RULES:
The variable name Manager MUST be used.
Manager MUST be declared ONLY ONCE per function.
If Manager already exists in the current function, you MUST reuse it.
You MUST NOT redeclare or shadow Manager.
If multiple modules are used, they MUST all use the same Manager instance.
FORBIDDEN:
Declaring auto& Manager = ... more than once in a function
Using different manager variables or calling Instance() repeatedly

6. Only modules provided by ModuleCodeGenerator may be used:
- You will receive a `modules` array; ONLY these modules are allowed
- Module class names must be inferred as F<ClassName>
- All module calls MUST be null-checked

7. Output format requirements:
- Final output MUST be strict JSON, for example:

```
{
  "objects": [
    {
      "class_name": "AChestInteractiveObject",
      "header_code": "Complete .h file code",
      "cpp_code": "Complete .cpp file code"
    }
  ]
}
```

- JSON strings must contain ONLY code text
- NO natural language outside JSON
- C++ comments inside code are allowed

8. Header/source style example (MUST FOLLOW):
......
Additional constraints:
- If linked_modules contains multiple modules, choose a reasonable call order based on interaction_flow.
- At minimum, include a Manager/GetModule example for the FIRST linked module, if it exists in ModuleCodeGenerator.modules.
- If interaction_flow specifies a concrete method (e.g. FInventoryModule::AddItemFromContainer), call that method (assume callable) and null-check the module first.

REMEMBER:
ALL generated interactive object classes MUST include LLMEDITOR_API.
Missing the export macro is considered a critical error.

Figure 14: The prompt details of $\mathcal{G}enerate_{Int}$ in Section 3.6.

The Prompt Details of $\mathcal{G}enerate_{Test}$

Role: You are an expert in generating evaluation test instructions for playable UE5 game demos.
Input: You will receive four inputs:
1. Scene Description
2. Gameplay Description
3. Output of Interactive Object Code Generator (an objects list containing each interactive object's class name and code)
4. Output of Gameplay Code Generator (a modules list)

Task: Based on the above inputs—especially the interactive object definitions—automatically generate a "test instruction script" for evaluating the game demo.
The test instructions are used to drive an automated AI agent to perform the following actions inside the demo:
Available Actions:
1. move_to(object_name)
    - Parameter: interactive object name
    - Description: Move the AI agent near the specified object
2. interact(object_name)
    - Parameter: interactive object name
    - Description: Trigger the object's interaction logic (TriggerInteract → HandleInteraction)

Output Format: Output strict JSON with the following structure:
```
{
 "evaluation_instructions": [
  {
   "step_id": 1,
   "action": "move_to",
   "target": "ObjectName",
   "description": "The AI moves near the object"
  },
  {
   "step_id": 2,
   "action": "interact",
   "target": "ObjectName",
   "description": "The AI interacts with the object"
  }
 ]
}
```
Requirements:
1. Generate two instructions for each interactive object:
    - Step 1: move_to
    - Step 2: interact
2. step_id must start from 1 and increment sequentially.
3. "ObjectName" must be automatically derived from `"class_name"`:
    - Remove the prefix `A`
    - Examples:
        - AChestInteractiveObject → ChestInteractiveObject
        - AEnemyBot → EnemyBot
4. Do NOT generate any C++ / Blueprint / UE code. Only output JSON test instructions.
5. The JSON must be fully valid and must not contain any extra explanations, comments, or natural language outside the JSON structure.

Figure 15: The prompt details of $\mathcal{G}enerate_{Test}$ in Section 3.7.

The Prompt Details of Overall Game Evaluation

Role: You are an Unreal Engine game demo reviewer.
Input: You will receive seven inputs:
1. [User Prompt] This section contains the description of the game scene and gameplay system.
2. [PCG Build Log] Review the procedural content generation log carefully. Consider:
- Are nodes created and connected correctly?
- Does the log indicate a successful build?
- Are models, nodes, connections, and parameters correctly applied?
- Use the PCG log and scene image(s) as reference.

3. [Module Analysis] Review the module analysis. Consider:
- Whether modules are logically designed and structured.
- Whether the module analysis identifies key components correctly.

4. [Module Generated Code] Review the generated module code. Consider:
- Code correctness and readability.
- Whether the generated code reflects the module analysis.

5. [Interactive Object Analysis] Review the interactive object analysis. Consider:
- Are interactive objects present and correctly defined?
- Are interactions described clearly and logically?

6. [Interactive Object Generated Code]Review the generated interactive code. Consider:
- Whether the code implements the intended interactions correctly.
- Code quality and completeness.

7. [Compile Log]Review the compile log. Consider:
- Are there compilation errors or warnings?
- Are all modules successfully compiled?

Task: Evaluate the demo along THREE INDEPENDENT DIMENSIONS, each scored from 1 to 10:
Evaluate the demo along THREE INDEPENDENT DIMENSIONS, each scored from 1 to 10:
1. PCG Scene Quality
- Are nodes created and connected correctly?
- Does the PCG log indicate a successful build?
- Are models, nodes, connections, and parameters correctly applied?
- Use the PCG log and scene images as reference.

2. Playability & Interaction
- Are interactive objects present and functional?
- Do the play images indicate smooth gameplay?
- Are actions and cause-effect chains visible?
- Does the demo feel playable and coherent?

3. Overall Visual & Game Design
- Scene aesthetics, composition, and clarity
- Model placement and environment
- Cohesiveness with intended game design

FINAL SCORE COMPUTATION:
final_score = PCG Scene Quality * 0.35 + Playability & Interaction * 0.35 + Visual & Game Design * 0.30
IMPORTANT:
- All scores MUST differ if quality differs.
- Output exactly one decimal for the final score.
- Output STRICT JSON only.
- Provide a grade as a string (A, B, C, or D) and a brief evidence-based summary.

Output Format: Output strict JSON with the following structure:
```
{
 "pcg_score": 0.0,
 "playability_score": 0.0,
 "visual_score": 0.0,
 "final_score": 0.0
}
```

Figure 16: The prompt details of Overall Game Evaluation in Section 4.2.

The Prompt Details of PCG Graph Evaluation

Role: You are an Unreal Engine gameplay system reviewer.
Input: You will receive seven inputs:
1. [User Prompt] This section contains the description of the gameplay system or feature that the user wants to implement or evaluate. It includes the goals, objectives, and any specific behaviors expected from the system.
2. [Module Analysis] This section provides a detailed analysis of the module's responsibilities, how well it covers the user's intent, and whether its reasoning and design decisions are sound. Focus on clarity, completeness, and correctness.
3. [Module Generated Code] This section contains the actual code of the module being evaluated. Include details such as class definitions, functions, Unreal Engine patterns used, and overall code structure. Highlight any strengths or weaknesses in the implementation.
4. [Interactive Object Analysis] This section analyzes the interactive objects in the scene, such as items, actors, or environment elements. Describe how well they integrate with the module logic, whether their interactions are meaningful, and if they properly affect or reflect module state changes.
5. [Interactive Object Generated Code] This section contains the code related to interactive objects, including how they call module functions, handle events, and update state. Note the quality and correctness of the integration.
6. [Compile Log] This section includes the results of attempting to compile the module and interactive object code. Describe any errors, warnings, or successful compilation messages.
7. [Runtime Evaluation Log]If the code compiled successfully, this section contains the runtime evaluation results, including observed behavior, interactions, and system responses during gameplay testing.
NOTE: IF Compilation FAILED,
You must account for this in your scoring.

- Module code is likely non-functional → module_code_score = 0
- Integration is severely impacted → integration_score should be low (e.g., 1)
- Analysis_score can still reflect your evaluation of reasoning quality

Task: You MUST evaluate the system along THREE INDEPENDENT DIMENSIONS.
1. Module Analysis Quality (1–10)
- Clarity of responsibilities
- Coverage of user intent
- Soundness of reasoning

2. Module Code Quality (1–10)
- Code structure and cleanliness
- Reasonable Unreal Engine patterns
- Clear state and API boundaries

3. Interactive Integration Quality (1–10)
- Interactive objects actually call module logic
- Calls are meaningful, not superficial
- Module state changes are driven by interaction
- Evidence of cause → effect chains

IMPORTANT:
- If compilation failed, enforce module_code_score = 0, integration_score = 0.
- Scores MUST differ unless quality is truly identical
- Do NOT default all scores to the same value

FINAL SCORE COMPUTATION (MANDATORY)
final_score = analysis_score * 0.30 + module_code_score * 0.30 + integration_score * 0.40
Output final_score with EXACTLY ONE DECIMAL
Do NOT round to integer
Output Format: Output strict JSON with the following structure:
```
{
  "analysis_score": "Score for module analysis, based on clarity, coverage, and reasoning.",
  "module_code_score": "Score for the module code quality, considering structure, patterns, and boundaries.",
  "integration_score": "Score for interactive object integration, based on meaningful interactions and state changes.",
  "final_score": "Weighted average of the three scores, calculated exactly as specified.“
}
```

Figure 17: The prompt details of PCG Graph Evaluation in Section 4.4.

The Prompt Details of Gameplay Code Evaluation

Role: You are a PCG (Procedural Content Generation) system evaluator.
Input: You will receive four inputs:
1. [User Prompt] This section describes the intended PCG content and goals, including what the generated graph or environment should look like, how complex it should be, and any specific design constraints.
2. [PCG BuildEnd Summary] This section contains the summary output from the PCG system's BuildEnd report. Include:
- Total number of nodes and how many were successfully created
- Total number of connections and their success/failure
- Parameters applied and any failures
- Any notable errors or warnings
- Overall generation statistics

Reported Status: This is the overall BuildEnd status reported by the system, e.g., "Success" or "Failure".NOTE: IF Compilation FAILED.
Task: You must evaluate the quality of the PCG graph generation based STRICTLY on the BuildEnd summary.
PRIMARY SIGNALS (MANDATORY)
Base your evaluation on the following:
- Nodes: total, created successfully, failed
- Connections: total, success, failed
- Parameters: total, applied successfully, failed

Node failures are MOST severe.
Connection failures are moderately severe.
Parameter failures are least severe.
FAILURE PENALTY RULE (HARD)
If the reported status is not "Success":
- Significantly reduce the score.

Apply conceptual penalty multipliers:
  - Minor failure → approx ×0.4
  - Moderate failure → approx ×0.3
  - Severe failure → approx ×0.2 or lower
- Non-success demos should usually score ≤ 7.9
- Severe failures should score much lower
- Different failing demos MUST NOT receive the same score

SUCCESS CASE SCORING (HARD BANDS)
If the reported status is "Success":
Place the score into EXACTLY ONE band based on graph scale and richness:
- 9.0–10.0 : Very large, dense graph with many nodes, many connections, and many parameters
- 8.5–8.9 : Medium-to-large graph, well connected, substantial parameter usage
- 8.0–8.4 : Small but non-trivial successful graph

Rules:
- Scores below 8.0 are NOT allowed for Success
- Not all Success demos may receive the same score
- Success demos of different scale MUST have different scores

ANTI-COLLAPSE RULE (CRITICAL)
You are FORBIDDEN from:
- Giving identical scores to all successful demos
- Reusing the same decimal score repeatedly
- Ignoring graph size differences

If multiple Success demos exist, their scores MUST form a meaningful ranking.
OUTPUT RULES
- Final score range: 1.0 – 10.0
- EXACTLY ONE decimal place

Output Format: Output strict JSON with the following structure:
```
{
  "pcg_score": "A numeric score based on the BuildEnd summary and rules above, with one decimal place.",
  "status": "The overall reported BuildEnd status, e.g., Success or Failure.",
  "analysis": "Brief, evidence-based explanation referencing nodes, connections, parameters, and any failures.“
}
```

Figure 18: The prompt details of PCG Graph Evaluation in Section 4.5.