VibeGame: Prompt-to-Game Development with AI-Native Engine and Self-Evolving Adversarial Agent Team

Wenbo Hu[1,*], **Ken Li**[1,*], **Jiazhe Wei**[1,*], **Yukang Cao**[2], **Weiyi Hong**[1], **Jiayi Dai**[1], **Chenjun Bai**[1], **Jiajun Liang**[1], **Yucheng Liao**[1], **Ruichuan An**[1], **Zeyu Lou**[1], **Haofan Wang**[4], **Wende Tan**[3], **Liucheng Guo**[3], **Yueming Lyu**[1], **Ziwei Liu**[2], **Chenyang Si**[1,†]

[1]PRLab, Nanjing University, [2]S-Lab, Nanyang Technological University, [3]Imperial College London, [4]Liblib.ai

[*]Equal Contribution, [†]Corresponding Author

Figure 1 Games created and edited by VibeGame from prompts, covering text-only creation, multimodal-input creation, IP transfer, genre transfer, and rule restructuring across diverse 2D game genres.

Abstract

Large language model agents have made it possible to create software through natural-language interaction, but extending this paradigm to complete game development remains challenging. A playable game project must keep code, scenes, assets, animations, physics, and configurations consistent, verify their combined behavior through interaction, and remain editable as user requirements evolve. We introduce VibeGame, a framework that creates and continuously edits complete 2D game projects from natural-language prompts and optional reference images. We first build an AI-native game engine that represents project state as structured text with explicit references and schema validation. Within this engine, we further enable frame-synchronous runtime control, allowing agents to inspect game states, execute actions, and verify behavior at their own pace. On this interface, we organize eight specialized agents into an Adversarial Agent Team for design, art, architecture, programming, auditing, play-testing, and review. By separating implementation from evaluation, the team converts failed structural, behavioral, and visual checks into repair tasks that re-enter development. Finally, we introduce a training-free self-evolution mechanism that distills accepted projects into reusable skeletons, verified modules, and collaboration contracts. We adapt and revalidate this experience when developing subsequent projects. Through qualitative demonstrations, we present game creation from text-only and multimodal inputs, together with continued editing through IP transfer, genre transfer, and rule restructuring. These cases illustrate how VibeGame connects project construction, interaction-grounded verification, and experience reuse in a continuous workflow from user intent to an editable and playable game project.

Date: August 6, 2026
Code: https://github.com/tettethu/VibeGame
Project Page: https://vibegame.tettet.org

1 Introduction

Among all forms of software, games are perhaps the most demanding to build. Rather than simply computing an output, a game must sustain an interactive world in real time, maintain a coherent visual identity, enforce consistent and balanced rules, and respond in ways that feel satisfying to the player. Building such an experience has traditionally required designers, programmers, and artists to collaborate for months or even years within professional engines such as Unity [1], Unreal Engine [2], and Godot [3]. The resulting cost and complexity place game development beyond the reach of many creators, leaving countless playable ideas unrealized.

AI has begun to lower this barrier from two directions, yet neither fully supports practical game creation. World models such as Genie [4] generate interactive experiences directly from video, shortening the distance from a concept to a playable prototype. Their outputs, however, remain implicit neural representations rather than editable engineering artifacts: they provide no source code to inspect, no project structure to version, and no dependable mechanism for making precise changes. Another line integrates large language models (LLMs) into commercial engines to automate parts of the production pipeline, as DreamGarden [5] does for Unreal. These engines, however, were designed around human-operated graphical workflows, with project state distributed across interfaces, configuration systems, and binary assets that agents cannot consistently inspect or modify. Consequently, neither direction produces what creators ultimately need: a complete game project that can be built, played, inspected, versioned, and iteratively revised.

Meanwhile, LLM-based coding agents [6, 7] have made "vibe coding" increasingly accessible, allowing non-programmers to describe software in natural language and refine it through iterative feedback. Games are the

natural next step for this paradigm, and the gap between the two directions above marks where it must land. We formulate this challenge as **Prompt-to-Game Development**: given a natural language description and optional reference images, the system must produce a complete playable game project, and support follow-up requests that continuously modify it.

Recent systems and benchmarks move closer to this goal. AutoUE [8] coordinates multiple agents to construct and test games in Unreal Engine; OpenGame [9] develops web games in an end-to-end manner and accumulates reusable templates and debugging experiences; and GameDevBench [10] and GameCraft-Bench [11] further evaluate whether coding agents can complete game-development tasks and produce interactively verifiable games. These efforts show that agents can construct and test increasingly complete games. However, most existing systems and benchmarks treat the first playable version as the endpoint of the task. In practical development, it is only the beginning: users would continue to request new mechanics, visual changes, balance adjustments, and bug fixes after the initial game has been delivered. Fulfilling these requests requires an agent to understand and extend an existing project while preserving the functionality and design choices that should remain unchanged.

This shift from one-time generation to continued development introduces three tightly coupled requirements. First, an **agent-operable project state** must expose the structure and dependencies among scripts, scenes, assets, animations, physics, and configurations. Direct file access alone is insufficient, as a seemingly local edit may violate hidden relationships elsewhere in the project. Second, **interaction-grounded verification** is equally important. A project can remain structurally valid yet fail at runtime when player input, physics, animation, and game logic interact. Because game execution unfolds much faster than an agent can inspect and reason about it, reliable verification requires controllable execution that can be paused, advanced, and examined beyond ordinary real-time play. Third, the system must **preserve validated development experience**. The initial prompt leaves many design and implementation choices unspecified, requiring different roles to establish and share decisions throughout development. Once validated, reusable project structures, features, and coordination procedures should inform subsequent work, while unverified or project-specific content must remain isolated. Together, these requirements enable an agent to modify a game continuously without losing control over its structure, behavior, or accumulated design intent.

To address these difficulties together, we introduce **VibeGame**, a unified framework that connects project construction, runtime verification, and experience reuse within a continuous development loop, as illustrated in Fig. 2. **(1)** At its foundation, an **AI-native game engine** is built to expose scenes, nodes, assets, scripts, and configurations as structured text with explicit references and schema validations. Frame-synchronous runtime control further allows agents to inspect game states, inject actions, and observe their behavioral consequences under controllable execution. **(2)** Built upon this shared interface, an **Adversarial Agent Team (AAT)** coordinates design, art, architecture, programming, auditing, and play-testing around a common project specification. Implementation roles construct the game, whereas auditor, player, and reviewer roles challenge it from structural, behavioral, and project-level perspectives. Failed checks are converted into repair tasks, forming an iterative loop that continues until the project satisfies the intended design and functionality. **(3)** Once the project is accepted, a **training-free self-evolution mechanism** distills its reusable experience into project skeletons[1], verified modules[2], and collaboration contracts[3]. Subsequent projects adapt and revalidate these artifacts rather than reusing them blindly. As a result, each completed game not only fulfills the current request but also provides a stronger and more reliable foundation for future development. Fig. 1 presents some examples produced by VibeGame.

[1]Skeleton is a runnable project template associated with a particular game type.
[2]Module is an encapsulated node script that implements functionality reusable across different game types.
[3]Contract is a document that adapts the general AAT workflow to a recurring feature or development pattern.

In summary, our contributions are three-fold:

- We develop an **AI-native game engine** that makes both project state and runtime behavior directly operable by coding agents. A structured scene-node representation exposes cross-artifact dependencies through explicit references and schema validation, while frame-synchronous execution allows agents to inspect, control, and verify the running game at their own pace.
- We design an **Adversarial Agent Team** that coordinates specialized roles around a shared design specification while separating implementation from evaluation. Static auditing, runtime play-testing, and independent review provide complementary forms of evidence, and failed checks are converted into repair tasks that re-enter the development loop.
- We introduce a **training-free self-evolution mechanism** that distills accepted games into reusable project skeletons, verified modules, and collaboration contracts. These artifacts guide subsequent creation and revision tasks, but are adapted and revalidated for each new project rather than assumed to remain universally correct.

2 Related Work

Agentic software development and vibe coding frameworks. Recent coding agents are pushing LLM-based programming from isolated code generation toward repository-level development. SWE-Bench [12] and SWE-agent [13] have advanced evaluation and tooling for agentic software engineering in real code repositories, while systems such as Claude Code [6], Codex [7], Cursor [14], and OpenHands [15] further enable agents to read projects, modify files, run commands, and iteratively fix issues based on execution feedback. These agents increasingly adopt agentic reasoning strategies such as ReAct [16] to interleave planning with action. Concurrently, vibe coding frameworks such as Trellis [17] externalize requirements, design decisions, task lists, and project memory as persistent artifacts within the repository, reducing agents' dependence on single-session chat context. In multi-agent collaboration, ChatDev [18] and MetaGPT [19] have introduced role specialization into general software development. VibeGame inherits these directions but focuses on the additional problems posed by game projects: a game's project state is far more complex than typical software, requiring code, art assets, scene configurations, and runtime behavior to be jointly maintained, with correctness verifiable only by actually running the game.

AI agents in games. Games have long served as testing environments for AI agents, from board games [20] to open-ended virtual worlds [21]. A recent survey [22] provides a comprehensive roadmap for LLM applications in games. Voyager [23], Cradle [24], Lumine [25], and generative agent simulations [26] demonstrate that LLM agents can complete complex tasks in games through actual play and observation, showing that game runtimes serve as challenging environments for evaluating agent capabilities. VibeGame draws on this "verify through actual gameplay" approach, but with a different purpose: our agents play in order to discover development issues by running and observing the game, then return to the project to fix them.

World models and neural game generation. Another line of work attempts to bypass traditional game engines and directly generate interactive environments. The Genie series of world models [4], diffusion-based game engines [27], and multi-actor generative approaches [28] learn action-controllable environments from video data, shortening the distance from prompts, images, or sketches to playable experiences. These methods demonstrate the potential of neural interactive worlds, but their outputs are typically not the inspectable, versionable, debuggable, and reusable engineering artifacts of traditional game development.

Engine-integrated game generation. Recent work has begun to integrate LLMs and generative models into game engines and 3D creation tools. Sketch2Scene [29] generates interactive 3D game scenes from user sketches; UnrealLLM [30] uses LLM-powered procedural content generation in Unreal Engine; 3Dify [31] combines MCP and RAG to call Blender, Unity, and UE toolchains for 3D content generation; DreamGarden [5] uses LLM-driven planning to iteratively construct UE game environments from a single prompt; and other work generates Unity game templates from design documents [32]. In parallel, LLM-based game generation has been explored through evolution [33], direct code synthesis [34], and level generation [35]. GameDevBench [10] provides an evaluation framework for agentic game development focused on Godot, although no benchmark yet targets JavaScript-based end-to-end game generation. These efforts demonstrate that engine access is necessary for prompt-to-game development, but most operate around the GUI workflows of com-

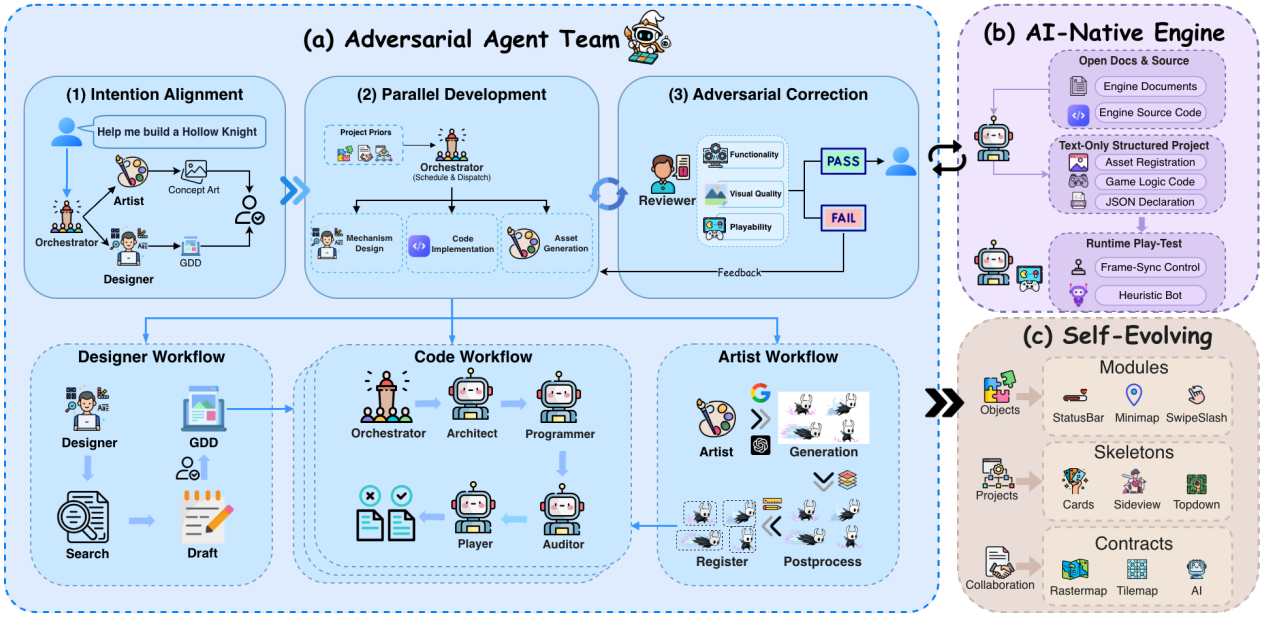

Figure 2 System overview of VibeGame. (a) The Adversarial Agent Team turns a user request into an accepted game through intention alignment, parallel development, and adversarial correction. (b) The AI-native game engine exposes a structured project representation and frame-synchronous runtime control for implementation and play-testing. (c) Training-free self-evolution extracts skeletons, modules, and collaboration contracts from accepted projects and returns them to future development as project priors.

mercial engines, where project state remains stored in binary assets and proprietary formats. By contrast, web-based 2D frameworks like Phaser [36] provide pure programming interfaces where projects can be fully expressed in text files, which makes them naturally suited for the operation of coding agents. VibeGame goes further: instead of simply connecting agents to an existing engine, it constructs an agent-oriented project representation, runtime control, and visual verification layer.

Summary. Existing vibe coding frameworks address requirement management and project memory for general software; game AI agents and world models demonstrate the potential of combining AI with interactive worlds; engine-integrated work shows that game engines are necessary tools for automated game generation. VibeGame sits at the intersection of these directions: it extends vibe coding's project-level workflows to game development while building an agent-oriented engine, multi-role review, and cross-project experience accumulation to address the unique challenges of maintaining code-asset consistency, verifying runtime behavior, and compensating for insufficient practice knowledge.

3 VibeGame: Integrated Game Development, Verification, and Reuse

VibeGame takes a natural-language request and optional reference images as inputs and produces a complete 2D game project that can be run, inspected, and modified through later requests. Fig. 2 presents the full project lifecycle and illustrates how the three core modules interact throughout development. We begin by tracing a complete run from the initial request to the delivered game, while establishing the terminology used in the remainder of the paper.

VibeGame is driven by a team of eight specialized agents that transform the user's request into a complete and validated game project. Among them, an *orchestrator*, which coordinates the agent team, would first clarify the intended game through follow-up questions when given the input. It then assigns the designer to produce a Game Design Document (GDD) and the artist to create concept art. After user confirmation, these artifacts establish the design ground truth for all subsequent development and evaluation. The orchestrator then organizes the project into three coordinated workflows: (1) the artist workflow creates and registers

visual assets, (2) the designer workflow refines and extends the GDD, and (3) the code workflow implements the game. For complex systems, the code workflow would further decompose the implementation into parallel sub-tasks, each handled by a four-agent pipeline in which the architect plans the solution, the programmer implements it, the auditor performs static checks, and the player evaluates the resulting behavior at runtime. Once all workflows are finished, an independent reviewer evaluates the complete project against the user-confirmed specification in terms of functionality, visual quality, and playability. Any failed checks are converted into repair tasks and returned to the corresponding workflows until the project is accepted.

Delivery does not terminate the development process. After a project is accepted, the orchestrator assigns agents to derive a reusable genre-level skeleton, extract self-checking modules that can transfer across projects, and distill collaboration contracts for features that require substantial coordination. These validated artifacts are stored in the experience repository and retrieved by future runs, allowing each completed project to improve the starting point for subsequent creation and revision tasks. Throughout this lifecycle, the AI-native engine provides the shared project representation and runtime interface on which all agents operate.

The following sections present the three modules in the order on which the development process depends: Sec. 3.1 details the AI-native engine, Sec. 3.2 the Adversarial Agent Team, and Sec. 3.3 the training-free self-evolution mechanism.

3.1 AI-Native Game Engine

A game engine provides the foundation on which a complete game project is constructed and executed. However, traditional engines are designed primarily for human-operated workflows and introduce three major barriers for coding agents. First, project state is often encoded through engine-specific identifiers or binary files, forcing agents to rely on graphical editors for inspection and modification. Second, debugging and runtime control are typically exposed through GUI-based tools that agents cannot operate reliably. Third, engine behavior may vary across versions, while documentation does not always capture the exact implementation of every feature.

We define a game engine as **AI-native** when it addresses these limitations through three properties. **(1) GUI-Independent**: the entire project is represented as plain-text files that agents can inspect and edit directly without graphical interaction. **(2) Runtime-Accessible**: agents can pause, step through, observe, and intervene in the running game, turning an asynchronous real-time system into a synchronously verifiable process. **(3) Source-Available**: agents can inspect the engine's uncompiled source code directly, rather than relying solely on documentation that may be incomplete or inconsistent with the implementation.

GUI-Independent. We build our engine on Phaser [36], a JavaScript-based framework for 2D games. While Phaser provides flexible low-level functionality, it imposes little structure on how a complete project should be organized, making agent-generated projects prone to fragmentation and difficult maintenance. We therefore introduce a structured, text-based project representation that agents can inspect and edit without a graphical interface. Specifically, each game is organized as a collection of scenes, with their configurations stored in type-checked JSON files that agents can edit directly. Each scene is further decomposed into a hierarchy of nodes, which serve as the basic building blocks of the project. A node groups its visual appearance, collision geometry, behavior scripts, and configuration into a single structured unit. These properties are declared explicitly in JSON fields rather than hidden inside constructors. Nodes shared across scenes are stored as standalone definition files and instantiated when needed at runtime. This scene-node-asset hierarchy gives every artifact a clear location and makes cross-file dependencies explicit. Schema validation can therefore detect invalid references, such as missing assets, before execution rather than allowing them to surface later as runtime failures. While agents operate entirely on the text representation, we also provide a graphical editor for human users to preview and adjust animations, nodes, and other project content.

Runtime-Accessible[4]**.** A structurally valid project may still fail when player input, physics, animation, and game logic interact during execution. Verifying these behaviors requires agents to control and inspect the running game rather than relying on static project files alone. We therefore expose the runtime through a

[4]Runtime refers to the running instance of a game, including its live scene state, rendered frame, physics state, and player inputs.

frame-synchronous programmatic interface. Agents can pause and resume execution, advance the game by a specified number of frames, inspect or modify runtime properties such as node positions and velocities, inject semantic actions such as "press jump", and capture screenshots. These controls allow agents to verify runtime behavior at their own pace through frame-by-frame inspection, scripted test sequences, or heuristic bots that react during real-time play.

Source-Available. Even with structured project files and runtime control, agents may encounter engine behavior that is not fully explained by the documentation, particularly when an interface has changed or a project uses uncommon functionality. To resolve such cases, we copy the engine's complete, uncompiled source code into the project directory during initialization. Agents can inspect the source at inference time to recover the exact APIs, assumptions, and implementation details needed for the current task, rather than relying solely on documentation that may be incomplete or inconsistent with the implementation.

3.2 Adversarial Agent Team

The engine provides the interfaces required to construct and test a game, but it does not determine what should be built, how development should be organized, or when the result is satisfactory. Assigning these responsibilities to a single agent would also require the same role to both produce and evaluate its own work, making behavioral failures and visual defects during gameplay easier to overlook. VibeGame therefore introduces an **Adversarial Agent Team** (AAT) comprising eight specialized roles: orchestrator, designer, artist, architect, programmer, auditor, player, and reviewer. We define each agent's responsibilities through role-specific prompts and execution hooks, and use persistent project artifacts to preserve decisions and communicate them across the team.

As shown in Fig. 2(a), the team organizes development into three stages: **Intention Alignment**, **Parallel Development**, and **Adversarial Correction**. Throughout this process, the orchestrator maintains the development state, coordinates the specialized roles, and converts failed checks or user feedback into new implementation and repair tasks.

3.2.1 Intention Alignment

Before development begins, the system must transform the user's high-level intent into an explicit and actionable design plan. Drawing on game-design principles and genre conventions, the designer expands requests such as "make a Hollow Knight-style boss fight" into a complete Game Design Document that defines gameplay mechanics, progression rules, and numerical systems. In parallel, the artist uses image-generation models to create concept art that establishes a consistent visual direction for subsequent assets. After user confirmation, the design document and concept art form the shared ground truth against which all downstream development and evaluation are conducted.

3.2.2 Parallel Development

Code Workflow. Under the orchestrator's coordination, code implementation is handled by a pipeline of 4 agents: architect, programmer, auditor, and player. First, the orchestrator decomposes game implementation into parallel components, retrieves relevant skeletons, modules, and collaboration contracts from the self-evolution repository (Sec. 3.3) as project priors, and prepares a Product Requirements Document (PRD) that defines the scope and acceptance criteria for each component. Based on the PRD and the current project state, the architect formulates a technical plan specifying both the implementation strategy and the corresponding verification procedure. After the orchestrator approves the plan, the programmer implements the required functionality within its constraints. The auditor then evaluates the resulting project statically using the engine's consistency-checking tools and verifies its compliance with the design specification. Finally, the player launches the game through the engine's runtime interface, tests its behavior under frame-synchronous control, and uses a vision model to assess whether the rendered result matches the intended appearance. When necessary, the player would adjust runtime parameters and write the validated values back to the project files. A sub-task is considered complete only after it passes both static auditing and runtime play-testing. We isolate parallel sub-tasks using Git worktrees before integrating their validated changes.

Artist Workflow. The artist agent converts visual requirements from the orchestrator into engine-ready assets. It invokes image-generation models, including gpt-image-2 and nano-banana-pro, to create the initial artwork, then applies post-processing operations such as background removal and size normalization. The processed assets are registered with the AI-native engine, and their paths, dimensions, and usage information are recorded for downstream development.

Designer Workflow. The designer translates gameplay requirements into explicit design specifications using game-design principles and genre knowledge. For each assigned task, it reviews relevant design patterns, drafts the required mechanics and numerical systems, and seeks user confirmation when the task introduces or changes a design decision. The approved content is then incorporated into the Game Design Document (GDD), which serves as the shared source of design guidance for all subsequent implementation and evaluation.

3.2.3 Adversarial Correction

After all sub-tasks have been completed, an independent reviewer acts as the final quality gate and evaluates the complete project along three dimensions: *functionality*, which checks whether the game behaves according to the design specification; *visual quality*, which examines asset completeness, stylistic consistency, and screen clarity; and *playability*, which assesses factors such as control responsiveness and the clarity of combat feedback. If the project fails any criterion, the reviewer rejects it with specific reasons, and the orchestrator converts the identified issues into new repair tasks. This build-review cycle continues until the project satisfies all acceptance criteria. The process runs automatically by default, but remains open to user intervention: the user may provide feedback at any time on game behavior, visual style, or design direction, and the orchestrator will incorporate this feedback into account in subsequent development.

3.3 Training-free Self-Evolution

The AI-native game engine and AAT jointly produce accepted game projects from text and image instructions. These accepted projects reflect both agent-side implementation and user feedback incorporated during development. VibeGame then uses a training-free self-evolution mechanism to distill reusable experience from these validated projects and use it to initialize future development runs. This reuse preserves the implicit knowledge accumulated during game development, which is difficult for general-purpose agents to preserve across independent development runs. We organize the resulting experience into three complementary forms: skeletons, modules, and contracts.

Skeletons. A skeleton is a runnable project template associated with a particular game type. It removes project-specific art assets but preserves the code architecture, tuning values, visual proportions, scene flow, and runtime structure of the original project. Each skeleton also includes error notes and an art pack. The error notes summarize recurring failures and their verified fixes, while the art pack records reusable visual-production recipes, including asset-generation prompts and post-processing conventions for the corresponding game type.

Modules. A module is an encapsulated node script that implements functionality reusable across different game types. In addition to the underlying behavior, a module may include self-checking logic that converts previously observed runtime failures into static validation rules executed before the game starts.

Contracts. A contract is a document that adapts the general AAT workflow to a recurring feature or development pattern. It specifies the role-specific inputs, outputs, dependencies, and verification criteria required to implement that feature, enabling agents to coordinate through a validated procedure rather than relying only on the generic build-and-review loop. For example, a map contract defines how the designer specifies map requirements, how the programmer represents the map in project files, how the artist supplies the required assets, and how the auditor and player verify the completed result.

4 Demonstrations

We demonstrate VibeGame's game development capabilities in two scenarios corresponding to the task defined in Sec. 1: creating a game from a new request (**Game Creation**) and continuing to edit an existing project

(**Game Edit**). Unless otherwise noted, the demonstrations use the automatic-by-default setting described in Sec. 3.2.3: after the user approves the request, the orchestrator continues development until reviewer acceptance, while the user may still interrupt the process with feedback. The prompt boxes in the figures report the final user-approved request rather than interface commands or intermediate clarification turns. In the default configuration, the orchestrator, artist, and architect use Claude Opus 4.7 [37]; the designer and programmer use Claude Sonnet 4.6 [38]; the auditor uses Claude Haiku 4.5 [39]; the player uses GPT-5.4-mini [40]; and the reviewer uses GPT-5.5 [41]. Individual demonstrations may vary slightly from this configuration. We also experimented with GLM [42] and MiniMax [43] for certain roles, but their lack of vision capabilities limited their effectiveness in tasks requiring visual judgment, such as play-testing and review.

4.1 Game Creation

Game creation starts from a text request and optional reference images and exercises the full loop from intention alignment to project delivery. In this paper, we report results for both text-only and multimodal requests.

4.1.1 Text Input

To test creation driven by a compact text request, we asked the system to build a Sekiro-style 2D boss fight: a retro pixel-art duel recreating the reed-field battle between Sekiro and Genichiro. The request specified the combat rules in a few sentences: the player can defend, perform frame-perfect deflections, attack, step-dodge, jump, slash in the air, and execute a deathblow once the boss posture is broken; the boss can defend and deflect, and special attacks are made readable through delayed wind-up frames.

The system started from an existing side-scrolling action skeleton, inheriting its scene structure and UI modules. However, Sekiro's core is a deflection-posture system entirely different from the skeleton's simple HP-based combat, requiring the system to build the combat layer from scratch: both sides have independent posture gauges, defense distinguishes regular blocking from frame-perfect deflection, boss attacks are divided into blockable normal attacks and perilous attacks requiring specific responses, and a full posture gauge triggers a multi-stage deathblow animation. The final output comprised 12 scripts, 2 character asset packs, 11 visual effect entities, and 1 UI asset pack. This case demonstrates how a short request that names a genre convention is expanded into a complete, correctly tuned combat system. The first row of Fig. 3 shows the request and resulting combat sequence.

4.1.2 Multimodal Input

To test multimodal creation, we provided a text request together with an author-created concept image for a Hollow Knight-style 1v1 boss fight. The text request specified the mechanical scope: the player has a basic attack, a charged attack, a dash, a jump, and an aerial attack, while the boss combines two melee skills with a ranged attack on a side-scrolling arena 1.5x the camera width. The concept image anchored the character designs, the ruined-cathedral arena, and the overall visual atmosphere, which the generated assets were required to match.

The second row of Fig. 3 showcases the multimodal input and resulting game. We can observe that the delivered slice realizes both sides of this specification. The boss gates its melee sweep, ground slam, and arcing projectile on the player's distance, keeps its preferred range, and can be staggered out of idle; the player's five actions cover the full movement-attack loop, including dash invincibility frames and aerial slashes; hit feedback combines screen shake, flash, and spark particles, with mask-style health icons and a full-width boss bar completing the HUD. All character and arena art was generated to match the concept image, preserving its silhouette language and cold palette. This case demonstrates how text can specify gameplay rules while the concept image anchors visual identity.

Figure 3 Game creation demonstrations. Top: Pixel Sekiro, a pixel-art boss fight created from a text prompt. Bottom: Hollow Knight boss slice, a side-scrolling boss fight created from a gameplay prompt and a concept image.

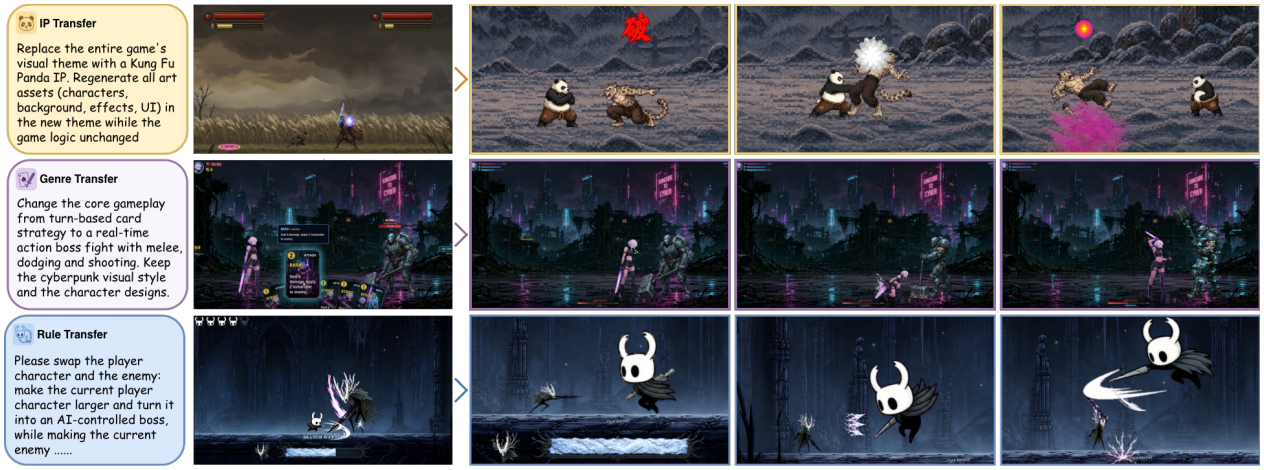

Figure 4 Game edit demonstrations. Top: IP transfer, replacing the visual theme while preserving gameplay logic. Middle: genre transfer, converting a turn-based card game into a real-time boss fight while preserving its visual identity. Bottom: rule transfer, swapping the player and boss roles while preserving their abilities and animations.

4.2 Game Edit

The next cases begin from accepted projects and alter different layers of the existing game. Specifically, in this paper, we report three types of editing: (1) IP transfer that changes visual assets while preserving mechanics; (2) genre transfer that replaces the central gameplay while preserving visual identity; and (3) rule transfer that changes the roles of existing characters while reusing their abilities and animations.

4.2.1 IP Transfer

Starting from a completed side-scrolling boss fight project, we ask the system to replace the entire game's visual theme with a Kung Fu Panda IP. The system regenerates all art assets (including characters, effects, UI, and background), and then adjusts configurations for the new sprite size and frame counts. The delivered project retains the original combat logic, illustrating an edit concentrated on the asset and configuration layers. See top row of Fig. 4. This validates the code-asset decoupling in the engine: when the project structure is well designed, IP transfer is accomplished primarily through asset replacement, with no need to rewrite core logic.

4.2.2 Genre Transfer

Starting from a completed turn-based card game project, we ask the system to change the core gameplay from card strategy to a real-time action boss fight. This is a deep transformation: the turn-based card-

playing mechanic was completely replaced with real-time melee, dodging, and shooting, while the system preserved the original project's cyberpunk visual style and character designs. Unlike IP transfer where only assets change, genre transfer requires the system to judge what can be preserved and what must be rewritten, demonstrating the system's ability to perform selective deep modification while maintaining overall consistency (Fig. 4, middle row).

4.2.3 Rule Transfer

Starting from the Hollow Knight-style boss fight created from multimodal input in Sec. 4.1.2, we tested the system's ability to handle unconventional creative requests by asking it to swap the player and boss roles, requiring only that each character keep its existing abilities and animations. In that source game, the player controlled a small horned warrior, while the antlered boss moved autonomously with a melee slam, a lunge, and a ranged thorn volley. The edit enlarges the former player character into an AI-driven boss and shrinks the former boss into the player-controlled character, so the player now wields the former boss's moveset while the former player character fights autonomously with its own dash, charged slash, and aerial attack. This request is not a simple asset exchange: the system must convert the former boss AI into a player control scheme, turn the former player's manual controls into autonomous boss AI, rebind inputs, and remap the HUD and win/lose flow, all without adding any new art or moves. Our system completed this transformation (Fig. 4, bottom row), demonstrating that it can restructure gameplay roles in response to non-standard edit requests.

4.3 Limitations

While our method could deliver impressive results across various tasks and situations, we do recognize certain limitations. Firstly, our study focuses on system design and illustrates VibeGame's capabilities through a diverse set of demonstrations, but does not provide a controlled quantitative evaluation. A comprehensive benchmark should assess initial generation quality, regression-free revision, visual consistency, and interaction quality, together with ablation studies that isolate the contribution of each component. We leave this evaluation to future work.

Secondly, the current engine supports only 2D web games. Extending the framework to 3D introduces additional challenges, including spatial physics, camera control, spatial audio, and more complex scene representations. It remains unclear whether a 3D project can retain the same fully text-based representation and whether its runtime can be exposed through equally reliable programmatic control.

Last but not least, current verification also remains imperfect. Even with vision model-assisted verification, judgments about visuals and manipulation can be wrong, and the quality of any single output remains probabilistic, requiring iteration as a safety net. The self-evolution mechanism also carries a contamination risk: once incorrect experience is adopted, it affects subsequent projects, making the user confirmation step essential.

5 Conclusion

We present VibeGame for creating and continuously revising complete game projects from natural-language and multimodal requests. Our system addresses three coupled challenges specific to prompt-to-game development: maintaining an agent-operable project state, verifying behavior through interaction, and preserving validated development experiences. Specifically, we first design an AI-native game engine which makes the project state fully readable and writable and the runtime fully controllable. We further propose an adversarial agent team that separates generation from review, continuously correcting errors through static audit, runtime testing, and visual assessment. Last but not least, we introduce a training-free self-evolution mechanism to distill reusable experience from each successful project. Through demonstrations spanning action boss fights, top-down shooters, IP transfer, genre transfer, and rule restructuring, we showcase that the system can produce and iteratively modify playable 2D games across diverse genres and scenarios. Future work will add controlled quantitative evaluation and explore extensions to 3D and additional game types.

References

[1] Unity Technologies. Unity game engine. https://unity.com, 2005. Accessed: 2026-06-11.

[2] Epic Games. Unreal engine. https://www.unrealengine.com, 1998. Accessed: 2026-06-11.

[3] Godot Engine contributors. Godot engine. https://godotengine.org, 2014. Accessed: 2026-06-11.

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

[5] Sam Earle, Dipesh Parajuli, and Andrzej Banburski-Fahey. DreamGarden: A designer assistant for growing games from a single prompt. *arXiv preprint arXiv:2410.01791*, 2024.

[6] Anthropic. Claude code: An agentic coding tool. https://github.com/anthropics/claude-code, 2025. Accessed: 2026-06-11.

[7] OpenAI. Codex: A lightweight coding agent. https://github.com/openai/codex, 2025. Accessed: 2026-06-11.

[8] Lei Yin, Wentao Cheng, Zhida Qin, Tianyu Huang, Yidong Li, and Gangyi Ding. AutoUE: Automated generation of 3d games in unreal engine via multi-agent systems. *arXiv preprint arXiv:2603.07106*, 2026.

[9] Yilei Jiang, Jinyuan Hu, Qianyin Xiao, Yaozhi Zheng, Ruize Ma, Kaituo Feng, Jiaming Han, Tianshuo Peng, Kaixuan Fan, Manyuan Zhang, and Xiangyu Yue. OpenGame: Open agentic coding for games. *arXiv preprint arXiv:2604.18394*, 2026.

[10] Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. GameDevBench: Evaluating agentic capabilities through game development, 2026.

[11] Tongxu Luo, Rongsheng Wang, Jiaxi Bi, Chenming Xu, Zhengyang Tang, et al. GameCraft-Bench: Can agents build playable games end-to-end in a real game engine? *arXiv preprint arXiv:2606.17861*, 2026.

[12] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? *arXiv preprint arXiv:2310.06770*, 2024.

[13] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Liber, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[14] Anysphere. Cursor: The ai code editor. https://www.cursor.com, 2024. Accessed: 2026-06-11.

[15] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, et al. OpenHands: An open platform for AI software developers as generalist agents, 2025.

[16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[17] Mindfold AI. Trellis: A spec-driven workflow framework for AI coding agents. https://github.com/mindfold-ai/Trellis, 2025. Accessed: 2026-06-11.

[18] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[19] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[20] Meta Fundamental AI Research Diplomacy Team (FAIR), Anton Bakhtin, Noam Brown, et al. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, 2022.

[21] Adrian Bolton, Alexander Lerchner, Alexandra Cordell, et al. SIMA 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.

[22] Roberto Gallotta, Graham Todd, Marvin Zammit, Sam Earle, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Large language models and games: A survey and roadmap. *IEEE Transactions on Games*, 2024.

[23] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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

[25] Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, Yujia Qin, Bo An, Libin Liu, and Guang Shi. Lumine: An open recipe for building generalist agents in 3d open worlds, 2025.

[26] Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, et al. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–22, 2023.

[27] Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. Diffusion models are real-time game engines. *arXiv preprint arXiv:2408.14837*, 2024.

[28] Alexander Sasha Vezhnevets, Jayd Matyas, Logan Cross, et al. Multi-actor generative artificial intelligence as a game engine. *arXiv preprint arXiv:2507.08892*, 2025.

[29] Yongzhi Xu, Yonhon Ng, Yifu Wang, Inkyu Sa, Yunfei Duan, Yang Li, Pan Ji, and Hongdong Li. Sketch2Scene: Automatic generation of interactive 3D game scenes from user's casual sketches. *arXiv preprint arXiv:2408.04567*, 2024.

[30] Song Tang, Kaiyong Zhao, Lei Wang, Yuliang Li, Xuebo Liu, Junyi Zou, Qiang Wang, and Xiaowen Chu. 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, 2025.

[31] Shun-ichiro Hayashi, Daichi Mukunoki, Tetsuya Hoshino, Satoshi Ohshima, and Takahiro Katagiri. 3Dify: a framework for procedural 3D-CG generation assisted by LLMs using MCP and RAG. *arXiv preprint arXiv:2510.04536*, 2025.

[32] Amna Hassan. Automated Unity game template generation from GDDs via NLP and multi-modal LLMs. *arXiv preprint arXiv:2509.08847*, 2025.

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

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

[35] Shyam Sudhakaran, Miguel González-Duque, Claire Glanois, Matthias Freiberger, Elias Najarro, and Sebastian Risi. MarioGPT: Open-ended text2level generation through large language models, 2023.

[36] Richard Davey and Photon Storm. Phaser – a fast, fun and free open source HTML5 game framework. https://phaser.io, 2013. Accessed: 2026-06-11.

[37] Anthropic. Claude opus 4.7. https://www.anthropic.com/news/claude-opus-4-7, 2026. Accessed: 2026-06-11.

[38] Anthropic. Claude sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6, 2026. Accessed: 2026-06-11.

[39] Anthropic. Claude haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5, 2025. Accessed: 2026-06-11.

[40] OpenAI. Introducing GPT-5.4-mini and nano. https://openai.com/index/introducing-gpt-5-4-mini-and-nano/, 2026. Accessed: 2026-06-11.

[41] OpenAI. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026. Accessed: 2026-06-11.

[42] Zhipu AI. GLM-5.1. https://z.ai/blog/glm-5.1, 2025. Accessed: 2026-06-11.

[43] MiniMax. MiniMax-M2.5 technical report. https://www.minimaxi.com/, 2025. Accessed: 2026-06-11.

Appendix

A More Demonstrations

Beyond the cases analyzed in Section 4, we used VibeGame to build a broader range of games spanning multiple genres and input modalities. Each game was produced from a natural language prompt, some accompanied by a reference image, exercising the same pipeline described in the main text. Together they illustrate the breadth of game types the system handles, from real-time action and arcade games to turn-based strategy and an LLM-driven social-deduction game.

Fruit Ninja

A fruit-slicing arcade game built from a text-only request. Each fruit splits into two physically simulated halves with fruit-specific weight, bounce, and juice effects; the game includes pointer-driven slicing, combo scoring, missed-fruit indicators, and a game-over state.

Figure 5 A three-fruit combo sliced mid-air with split halves and juice effects.

KOF

An arcade fighting game generated from a single-sentence prompt requesting a King of Fighters remake, with no reference image. Two fighters trade combos under health bars, a round timer, and a neon-lit stage.

Figure 6 Two fighters trade blows with an impact spark on the neon stage.

Dungeon Roguelike

A Soul Knight-style dungeon-shooter roguelike. The map is built from rooms and corridors; entering a room seals the doors and starts combat. The player picks up weapons from chests, carries up to two at a time, and aims toward the mouse, with health, shield, and mana as stats.

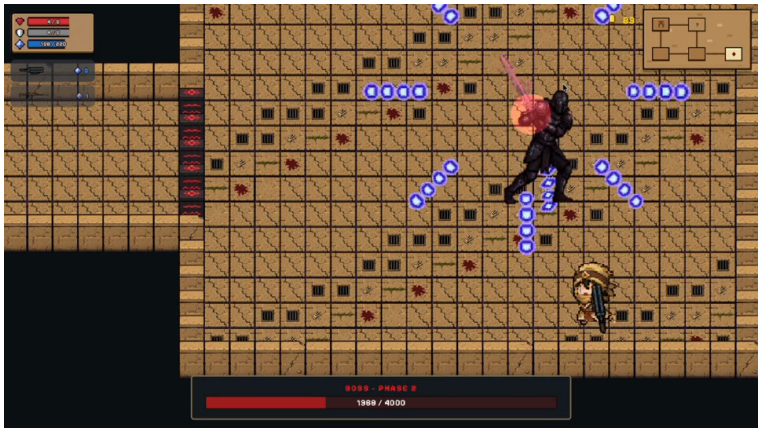

Figure 7 The player fires a spread of projectiles at an enemy inside a sealed combat room.

Parkour Platformer

A cartoon parkour platformer. The protagonist runs and dashes across a field of floating balloons; stepping on a balloon pops it and launches her upward, and the level is cleared by chaining precise balloon landings.

Figure 8 The protagonist floats above a balloon, with the goal flag on its platform.

Civilization prototype

A simplified Civilization-style 4X strategy game on a hex map, rendered in a cartoon hand-drawn style with city management and unit combat. It runs on a separated frontend and backend with multiplayer support.

Figure 9 A selected warrior unit and its stats beside a city on the hex map.

AI Werewolf

A single-player Werewolf game driven by the engine's built-in LLM module. Eleven AI players fill a 12-role setup (villagers, werewolves, seer, hunter, witch, idiot), conducting sheriff elections, day discussions, and night actions; each AI's speech is rendered as on-screen text.

Figure 10 The round-table layout with AI player portraits and live speech text during the sheriff election.

HK to Demon Slayer

An IP transfer applied to the Hollow Knight-style boss-fight prototype from Section 4.1.2: the system restyled the entire game into a Demon Slayer theme, recreating the Mugen Train battle between Tanjiro and Upper Rank Three while keeping the underlying combat logic unchanged.

Figure 11 Tanjiro faces Upper Rank Three under the moon on the Mugen Train night field.