OpenGame: Open Agentic Coding for Games

Anonymous ACL submission

Abstract

Game development sits at the intersection of creative design and intricate software engineering, demanding the joint orchestration of game engines, real-time loops, and tightly coupled state across many files. While Large Language Models (LLMs) and code agents now solve isolated programming tasks with ease, they consistently stumble when asked to produce a fully playable game from a high-level design, collapsing under cross-file inconsistencies, broken scene wiring, and logical incoherence. We bridge this gap with **OpenGame**, the first open-source agentic framework explicitly designed for end-to-end web game creation. At its core lies **Game Skill**, a reusable, evolving capability composed of a *Template Skill* that grows a library of project skeletons from experience and a *Debug Skill* that maintains a living protocol of verified fixes-together enabling the agent to scaffold stable architectures and systematically repair integration errors rather than patch isolated syntax bugs. Powering this framework is GameCoder-27B, a code LLM specialized for game engine mastery through a three-stage pipeline of continual pre-training, supervised fine-tuning, and execution-grounded reinforcement learning. Since verifying interactive playability is fundamentally harder than checking static code, we further introduce OpenGame-Bench, an evaluation pipeline that scores agentic game generation along Build Health, Visual Usability, and Intent Alignment via headless browser execution and VLM judging. Across 150 diverse game prompts, OpenGame establishes a new state-of-the-art. We hope OpenGame pushes code agents beyond discrete software engineering problems and toward building complex, interactive real-world applications. Our framework will be fully open-sourced.

1 Introduction

Video games represent one of the sharpest challenges in automated software engineering, demanding a rare fusion of rigorous logic, aesthetic design, and interactive storytelling. Unlike traditional utility software, a playable game is a real-time system whose quality depends on the seamless orchestration of update loops, physics, event handling, asset pipelines, and tightly coupled state across many files. This makes game creation both technically demanding and creatively expensive. Although democratizing game development has long been a goal of the broader creative community, the barrier to entry remains stubbornly high: turning an idea into a playable artifact still requires simultaneous mastery of engine architectures, programming languages, and fragile systems integration.

The recent surge in Large Language Models (LLMs) and autonomous code agents has transformed the landscape of software engineering (Jimenez et al., 2024; Cognition AI, 2024). Modern agents can solve discrete algorithmic problems, generate boilerplate, and even navigate mature repositories with impressive competence. Yet when tasked with end-to-end game creation, these general-purpose systems hit a formidable "complexity wall." Generating a calculator script or an isolated gameplay mechanic is far easier than constructing a coherent, fully playable game. In practice, we observe three recurring failure modes in frontier models: (1) **Logical Incoherence**: the model loses track of global state across the game loop, producing projects that freeze, fail to terminate, or never realize key mechanics; (2) **Engine-Specific Knowledge Gaps**: general models often ignore or misuse engine abstractions, re-implementing mechanics from scratch instead of correctly leveraging framework-native physics, scene, and event systems; and (3) **Cross-File Inconsistencies**: even when individual files look plausible, the overall project frequently breaks due to mismatched asset keys, flawed scene wiring, missing configuration fields, or broken initialization order. These failure modes are precisely what prevent

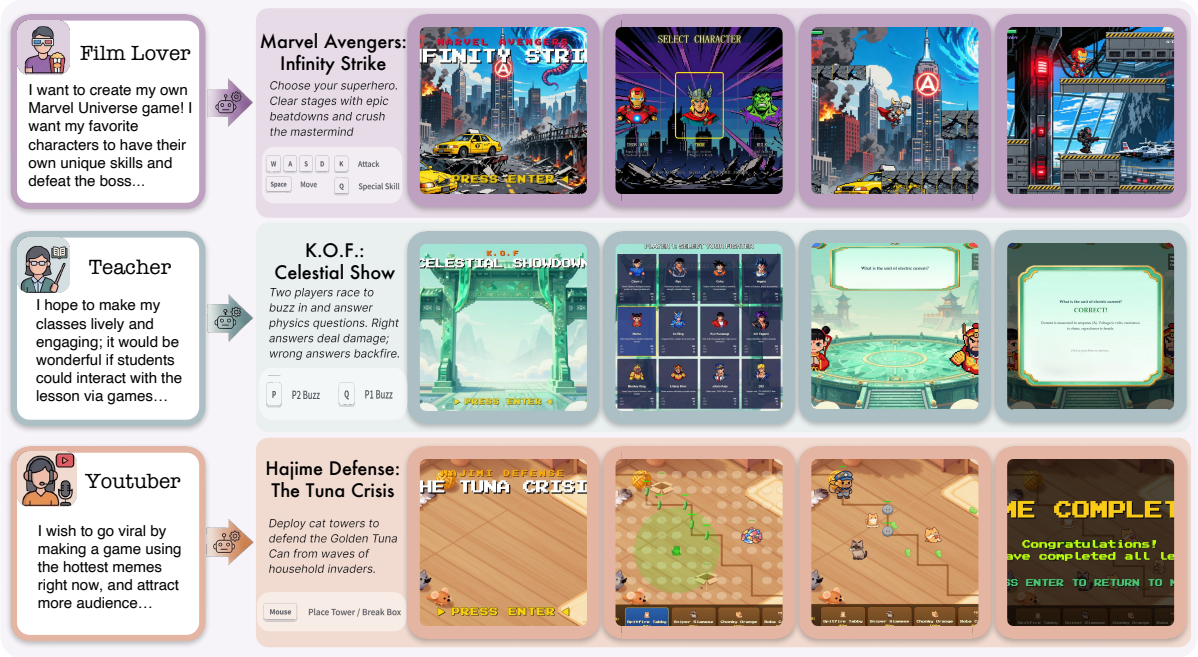

Figure 1: **End-to-end agentic game generation with OpenGame.** Diverse users provide natural language specifications to autonomously create fully playable 2D games across distinct genres (e.g., action, educational, and tower defense). Each generated project features a complete game lifecycle seamlessly integrated with multimodal visual and audio assets.

natural-language game design from being reliably brought to life.

To bridge this gap, we argue that the field must move beyond generalist code agents toward specialist frameworks that understand the intrinsic structure of games. We therefore present **OpenGame**, the first open-source agentic framework explicitly designed for end-to-end web game creation. At the core of OpenGame is **Game Skill**, a reusable capability for translating a natural-language design specification into a runnable project. Game Skill addresses systemic integration failures through two evolving components. First, **Template Skill** grows an evolving library of project skeletons ($\mathcal{L}$), starting from a single game-agnostic meta template ($\mathcal{M}_0$) and expanding into specialized template families such as gravity-based side view and top-down continuous motion. This sharply reduces the search space of generation and stabilizes project-wide structure. Second, **Debug Skill** maintains a living debugging protocol ($\mathcal{P}$) updated from observed build, test, and runtime outcomes, allowing the agent to accumulate verified fixes and systematically resolve high-frequency integration failures rather than repeatedly rediscovering them from scratch.

Supporting this framework is a domain-specialized foundation model, **GameCoder-27B**. Rather than relying solely on prompting a general code model, we train GameCoder-27B through a three-stage pipeline of continual pre-training, supervised fine-tuning, and execution-grounded reinforcement learning. This pipeline equips the model with engine-specific architectural priors, API usage patterns, and the logical discipline required for multi-file gameplay systems, providing a stronger substrate for the downstream agent.

Finally, progress in this area is bottlenecked by evaluation. Validating a game is fundamentally harder than verifying a standard function: code that compiles may still produce an unplayable, inert, or mechanically incoherent experience. Existing software benchmarks predominantly rely on static input-output unit tests, which are poorly suited to the temporal and interactive nature of gameplay. To address this gap, we introduce **OpenGame-Bench**, an evaluation pipeline designed to assess whether an agent can actually build interactive web games. OpenGame-Bench moves verification from static code analysis to dynamic playability assessment, scoring generated projects along build correctness, visual usability, and intent satisfaction through headless browser execution and multimodal judging.

In summary, our contributions are as follows:

- We propose OpenGame, the first open-source, tool-augmented coding agent dedicated to generating playable web games from natural-language specifications, enabling creative design ideas to be brought to life as executable artifacts. Central to this framework is Game Skill, an evolving combination of Template Skill and Debug Skill that stabilizes project scaffolding and resolves recurrent cross-file failures.

- We train GameCoder-27B, a domain-specialized code model through continual pre-training, supervised fine-tuning, and execution-grounded reinforcement learning to better master game engine patterns, API usage, and complex gameplay logic.

- We introduce OpenGame-Bench, a new evaluation paradigm for interactive code generation, moving beyond static unit tests to measure build health, visual usability, and intent alignment for end-to-end web game creation.

2 Related Work

Agentic Benchmarks and Software Development. SWE-Bench (Jimenez et al., 2024; Yang et al., 2024a) catalyzed agentic software engineering by shifting evaluation from isolated functions to repository-level issues, and subsequent benchmarks have extended this paradigm to complex unimodal reasoning (Chan et al., 2025; Merrill et al., 2026; Yang et al., 2025), multimodal frontend development (Zhu et al., 2025; Si et al., 2025; Yang et al., 2024b), computer use (Xie et al., 2024), and web navigation (Zhou et al., 2024; Koh et al., 2024). More recently, GameDevBench frames game development itself as an agentic testbed (Chi et al., 2026). Game development uniquely bridges two regimes—the multimodal grounding of computer-use agents and the structural code synthesis of software agents—and **OpenGame-Bench** targets this intersection by evaluating end-to-end interactive web game construction with dynamic playability rather than static task completion.

AI in Games: From Playing to Content Generation. Games have long served as testbeds for AI intelligence (Gallotta et al., 2024), from Deep Blue (Campbell et al., 2002) to modern generalists like SIMA 2 (SIMA Team, 2025) and LLM agents that play Pokémon (Karten et al., 2026, 2025; Comanici et al., 2025; Nunu AI, 2024). On the creation side, prior work has progressed from procedural content generation (Summerville et al., 2018; Farrokhi Maleki and Zhao, 2024) and evolutionary level design (Sudhakaran et al., 2023) to systems that bypass engines entirely with LLM-driven stories (Vezhnevets et al., 2023, 2025) or neural world models that simulate frames directly (Bruce et al., 2024). Driving the actual game *development* process, however, remains an open structural challenge.

Structural Game Engineering and Web-Based Frameworks. Industry-standard engines such as Unreal (Epic Games, 1998) and Unity (Unity Technologies, 2005) rely on proprietary GUIs and binary asset pipelines that are difficult for text-based agents to manipulate. In contrast, web 2D frameworks like Phaser (Davey and Photon Storm, 2013) expose a purely programmatic API in raw JavaScript or TypeScript, making them an ideal substrate for agentic software engineering. **OpenGame** exploits this by combining GameCoder-27B with a **Game Skill**—an evolving *Template Skill* for project scaffolding and a living *Debug Skill* for cross-file repair—to produce verifiable, executable web games, while `OpenGame-Bench` provides the matching dynamic playability evaluation.

3 Methodology

OpenGame is built from the interaction of a domain-specialized code model and a structured multimodal coding agent. Our methodology has three pillars: the training pipeline of the base model (GameCoder-27B), the design of the autonomous game-generation workflow, and the continual evolution of the agent through reusable game-development skills.

3.1 Base Model Training

We develop **GameCoder-27B** on top of a Qwen3.5-27B backbone via a three-stage pipeline. **Continual Pre-Training (CPT)** adapts the model to interactive web games using a corpus of open-source Phaser and JavaScript/TypeScript game repositories, official documentation, and community tutorials, establishing a prior over game loops, physics, asset usage, and state management. **Supervised Fine-Tuning (SFT)** aligns the model with game-design instruction-following on a synthetic QA dataset whose prompts are authored by *gpt-codex5.1* and whose target solutions are produced

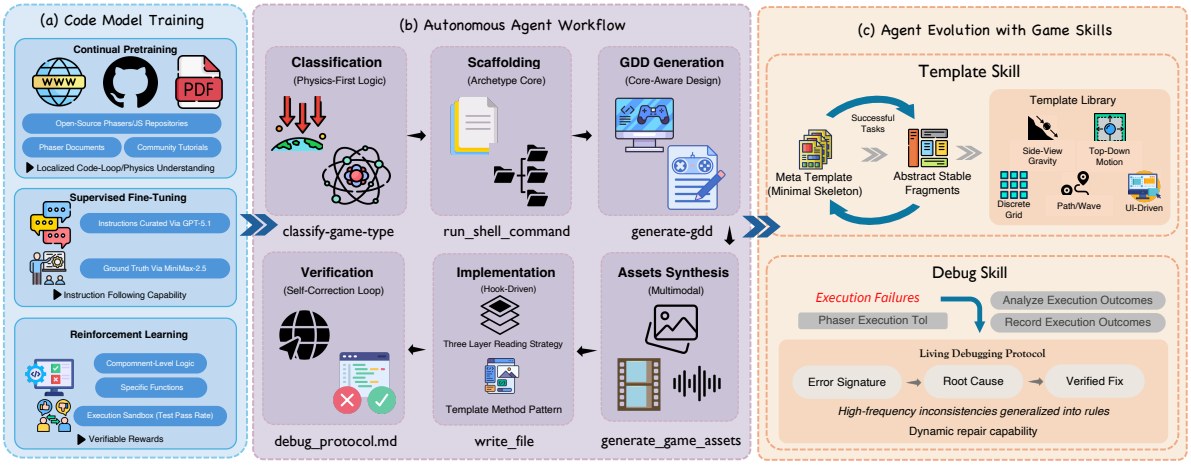

Figure 2: **The OpenGame architecture.** The framework integrates three coupled components: (a) a multi-stage code-model training pipeline that establishes engine-specific priors, (b) an autonomous agent workflow that translates natural-language game ideas into runnable projects through a structured six-phase process, and (c) an agent-evolution module that continuously refines structural scaffolding (Template Skill) and repair behavior (Debug Skill) through accumulated experience.

by *minimax2.5*. **Reinforcement Learning (RL)** then refines the model with execution-based feedback at the component level: rather than generating whole games, the model synthesizes single-file gameplay modules (e.g., collision, state machines) that are scored against unit tests, with the reward combining build success and aggregate test pass rate. This grounds the model in deterministic, executable logic before the downstream agent assembles components into a multi-file project. Full corpus statistics and training details are reported in Appendix C.

3.2 Code Agent Design

To prevent the logic hallucination, context drift, and brittle integration typical of naive end-to-end generation, OpenGame orchestrates the agent through six operational phases—classification, scaffolding, design generation, asset synthesis, code implementation, and verification—with persistent state tracked by a dedicated *todo_write* tool.

Classification, scaffolding, and assets. The workflow first invokes *classify-game-type*, which applies a **Physics-First Classification** rule that maps each request to one of five archetypes based on physical constraints rather than ambiguous genre labels (e.g., "falling without ground support" → *platformer*, "snapping to a grid" → *grid_logic*). The agent then scaffolds the project via *run_shell_command*, copying the shared core, the appropriate *modules/{archetype}* codebase, and architectural documentation into the workspace, and invokes *generate-gdd* to produce a technical Game Design Document whose mechanics are constrained by the archetype-specific API loaded from the scaffolded docs. Asset generation is then driven by *generate-game-assets* (backgrounds, character animations, static items, audio) and *generate-tilemap* (ASCII layouts → JSON tilemaps) against the GDD's asset registry; the produced *asset-pack.json* records exact texture and asset keys, substantially reducing downstream asset-reference hallucinations.

Implementation and verification. Before writing gameplay logic, the agent merges GDD parameters into *gameConfig.json* to enforce a data-driven design→code interface, and applies a **Three-Layer Reading Strategy** that progressively loads (1) an API summary, (2) the targeted source file (e.g., *_Template*.ts*), and (3) the implementation guide, ordered to maximize immediate salience. Code generation then follows a **Template Method Pattern**: the agent copies template files and overrides designated hook methods (e.g., *setupCustomCollisions*) rather than rewriting from scratch, preserving the deterministic lifecycle of the base classes. Finally, the agent enters a verification loop that first performs a static self-review against *debug_protocol.md*, then runs *npm run build* / *npm run test* under headless browser evaluation, parses compiler output on failure, and iteratively repairs the project until a playable game is produced. This protocol grounds the more general Debug Skill described next.

3.3 Agent Evolution with Game Skills

We equip the agent with **Game Skill**, a reusable capability for converting a natural-language game specification into a runnable project. Game Skill consists of two components: **Template Skill**, which stabilizes project structure, and **Debug Skill**, which improves reliability during verification and repair.

Problem setting. Given a user specification x describing mechanics, theme, and constraints, the agent must produce a project y that can be built and executed. In practice, failures are more often caused by cross-file inconsistencies—spanning assets, configuration, scene wiring, and initialization order—than by isolated syntax errors. Game Skill is designed to reduce these systemic failures while keeping generation stable across diverse requests.

Template Skill. The agent begins with a single **meta template** $\mathcal{M}_0$, a minimal game-agnostic project skeleton that defines the universal structure required for a playable game, including project layout, initialization, asset loading, scene loops, and configuration interfaces. $\mathcal{M}_0$ intentionally does not assume any genre, physics regime, or gameplay mechanic. As the agent completes more tasks, Template Skill maintains an evolving template library $\mathcal{L}$ through experience accumulation. After each task, the agent identifies code fragments that are (i) stable across games, (ii) broadly useful, and (iii) safe to reuse. These fragments are abstracted into reusable template units and constraints, then merged into $\mathcal{L}$. Over time, $\mathcal{L}$ grows from $\mathcal{M}_0$ into a compact set of **specialized template families** that reflect recurring physics and interaction regimes. In our setting, this process consistently yields five families: gravity-based side view, top-down continuous motion, discrete grid logic, path-and-wave dynamics, and UI-driven gameplay. Crucially, these families are not assumed a priori; they emerge from repeated reuse and robustness considerations. For a new request x, the agent selects an appropriate template family from $\mathcal{L}$ and instantiates it to obtain a stable project skeleton. Game-specific content is then introduced through a limited set of extension points while preserving the overall structure. This reduces the search space of code generation and improves cross-file consistency.

Debug Skill. Debug Skill targets the systematic failures that repeatedly appear in generated game projects. Rather than relying on a fixed hand-written checklist, the agent maintains a **living debugging protocol** $\mathcal{P}$ that is updated from observed build, test, and runtime outcomes. Concretely, each time a failure occurs, the agent records a structured entry containing an error signature, a root cause, and a verified fix. These entries are added to $\mathcal{P}$ and reused in future tasks. In addition, $\mathcal{P}$ includes lightweight pre-execution validations that target high-frequency inconsistency classes discovered previously, such as mismatched asset keys, missing configuration fields, or invalid scene transitions. When a failure pattern recurs, the protocol generalizes it into a reusable rule; when a novel failure appears, the protocol expands with a new entry. In this way, debugging knowledge becomes cumulative and persistent, improving reliability over time without increasing prompt complexity.

Overall execution. Template Skill provides a stable structural starting point, while Debug Skill verifies, diagnoses, and repairs the project until it becomes buildable and runnable, logging validated fixes back to the protocol (see Algorithm 1).

4 Evaluation

We evaluate OpenGame on a benchmark of 150 browser game tasks, measuring performance along three dimensions: build correctness, visual quality, and intent satisfaction. Because open-ended interactive software is difficult to assess with static checks alone, all experiments are conducted through **OpenGame-Bench**, our automated evaluation pipeline for dynamic game execution.

4.1 Experimental Setup

Benchmark. OpenGame-Bench consists of 150 tasks derived from 150 unique natural-language prompts spanning five game genres, including platformers, top-down shooters, puzzle games, arcade classics, and strategy. Each prompt is a self-contained game design specification used as the sole input; no reference implementation or starter code is provided. Tasks are primarily human-crafted by the authors and in-house annotators, complemented by a small set adapted from public game-jam repositories, and are manually verified to be technically achievable within 2D web frameworks.

Evaluation Protocol. For each task, the generated project directory is evaluated by OpenGame-Bench. A run is considered valid only if the project

builds successfully (when a build step is required), is served over a local HTTP server without fatal runtime errors, and produces at least one non-empty screenshot during automated play. Runs that fail any of these preconditions are reported separately as pipeline errors. To account for stochasticity, we evaluate each task three times with different random seeds and report mean scores.

Metrics. OpenGame-Bench scores each generated game on three dimensions, each scaled to $[0, 100]$. **Build Health (BH)** measures whether the project compiles, loads, and renders without critical errors. This captures a broad class of failures—broken dependencies, JavaScript runtime exceptions, and silent network failures—that a binary pass/fail criterion would collapse into a single outcome. **Visual Usability (VU)** combines a pixel-level heuristic (frame entropy and motion detection) with a Vision-Language Model (VLM) judge score, rewarding games that render coherent, animated, and visibly interactable content. **Intent Alignment (IA)** derives a weighted pass rate from per-requirement verdicts produced by a VLM judge against a structured requirement specification automatically compiled from the original prompt.

4.2 Baselines

We compare OpenGame against a broad suite of strong baselines spanning both direct LLM generation and established agentic frameworks. **Direct Code LLMs (Base Models).** To characterize zero-shot game-generation ability, we evaluate frontier models given the prompt together with the instruction to output Phaser 3 code files. These include *open-source* models (Qwen-3.5-Max (Qwen Team, 2026), MiniMax m2.5 (MiniMax, 2026), GLM-4.5 (Zhipu AI, 2025), Kimi K2.5 (Kimi Team, 2026), and DeepSeek V3.2 (DeepSeek-AI et al., 2025)) and *closed-source* models (Claude Sonnet 4.6 (Anthropic, 2026), GPT-5.1 (OpenAI, 2025), and Gemini 3.1 Pro (Google DeepMind, 2026)). **Agentic Frameworks.** To compare against existing multi-turn software-engineering systems, we evaluate two prominent frameworks: ***qwen-code*** (Qwen Team, 2025), paired with multiple backend models (Qwen-3.5-Max, MiniMax m2.5, Kimi K2.5, and Claude Sonnet 4.6) to isolate the effect of the underlying reasoning engine; and ***Cursor*** (Anysphere, 2024), evaluated with Kimi K2.5 and Claude Sonnet 4.6 backends.

4.3 Main Results

With Claude Sonnet 4.6 as the backbone, OpenGame sets a new state of the art on all three metrics (BH/VU/IA = $72.4/67.2/65.1$), beating the strongest baseline—Cursor with Claude Sonnet 4.6—by $+5.6/+5.8/+6.2$ points, with the largest gain on Intent Alignment, suggesting that our structured planning, template-based scaffolding, and iterative verification preserve user-specified mechanics rather than hallucinating engine behavior. The three metrics are only partially correlated across systems: visually tuned models (e.g., Gemini 3.1 Pro) lead on VU but lag on IA, while code-specialized models (e.g., DeepSeek V3.2) achieve strong BH but weak VU and IA, confirming that binary success rates would conflate qualitatively different failure modes. Our custom-trained *GameCoder-27B* backbone is also competitive: at $63.9/57.0/54.1$ it outperforms every direct LLM baseline on BH and IA, and edges out the much larger *qwen-code* (w/ Claude Sonnet 4.6) on BH ($+0.7$) and VU ($+2.7$) while trailing on IA (-3.7). Even so, the full system still leaves $\approx 34.9\%$ of weighted mechanical requirements unsatisfied, reflecting the difficulty of mapping ambiguous prompts to self-consistent, playable multi-file systems.

4.4 Ablation Studies

We ablate the three main pillars of OpenGame—the GameCoder-27B training pipeline, the agentic workflow, and the evolving Game Skills—to isolate each contribution.

Training pipeline. Table 2 shows that, even with the full OpenGame framework fixed, continued training on Qwen-3.5-27B yields incremental gains: CPT improves Build Health via better Phaser API familiarity, SFT contributes the largest Intent Alignment boost ($+1.9$), and execution-grounded RL further lifts Visual Usability and Intent Alignment, reaching the full GameCoder-27B score of $63.9/57.0/54.1$. The gains are incremental rather than transformative, indicating that most of OpenGame's headline improvement comes from the framework itself rather than the backbone alone.

Agent workflow. Table 3 isolates the workflow mechanisms with the backbone fixed. The **Template Method Pattern** (Hook-Driven Implementation) is the dominant constraint: forcing the agent to write scripts from scratch rather than overriding

Table 1: Performance evaluation on OpenGame-Bench. **Build Health (BH)** measures compilation and runtime stability; **Visual Usability (VU)** evaluates the rendering of coherent, interactable content; **Intent Alignment (IA)** scores the satisfaction of natural-language prompt requirements. Best results in **bold** and second best are underlined.

Category	System / Model	Build Health	Visual Usability	Intent Alignment
Direct LLMs (Open-Source)	Qwen-3.5-Max	51.8	35.5	38.9
	MiniMax m2.5	39.7	39.3	31.8
	GLM-4.5	46.5	45.0	31.2
	Kimi K2.5	45.6	46.8	44.6
	DeepSeek V3.2	57.0	38.9	33.5
Direct LLMs (Closed-Source)	Claude Sonnet 4.6	58.5	50.8	50.3
	GPT-5.1	57.4	52.9	49.4
	Gemini 3.1 Pro	53.6	60.2	42.1
Agentic Frameworks	*qwen-code* (w/ Qwen-3.5-Max)	57.7	41.3	40.2
	qwen-code (w/ MiniMax m2.5)	48.1	39.1	34.6
	qwen-code (w/ Kimi K2.5)	59.6	52.1	49.9
	qwen-code (w/ Claude Sonnet 4.6)	63.2	54.3	57.8
	Cursor (w/ Kimi K2.5)	57.1	55.2	54.2
	Cursor (w/ Claude Sonnet 4.6)	66.8	61.4	58.9
Ours (OpenGame)	*w/ Qwen-3.5-27B*	62.8	53.8	49.8
	w/ GameCoder-27B	63.9	57.0	54.1
	w/ Claude Sonnet 4.6	**72.4**	**67.2**	**65.1**

Table 2: Ablation of the GameCoder-27B training pipeline. All rows are evaluated with the same OpenGame agentic framework to isolate the incremental value of domain adaptation (CPT), instruction alignment (SFT), and execution-based RL on the Qwen-27B backbone.

Model Stage	Training Components	Build Health	Visual Usability	Intent Alignment
Base Model	Qwen-3.5-27B (in OpenGame)	62.8	53.8	49.8
Stage 1	+ CPT	63.2	54.7	50.6
Stage 2	+ CPT + SFT	63.5	55.7	52.5
Stage 3 (Full)	**+ CPT + SFT + RL**	**63.9**	**57.0**	**54.1**

base-class hooks drops BH by 10.1 and IA by 11.6, and frequently triggers fatal lifecycle errors. Disabling the **Three-Layer Reading Strategy** costs 8.6 IA, showing that progressive salience control is still required even with long contexts, while removing **Physics-First Classification** gives the smallest—but non-trivial—drop, mainly by routing tasks to mismatched template families.

Agent evolution. Finally, we decompose Game Skills into maturity stages of Template Skill ($\mathcal{L}$) and Debug Skill ($\mathcal{P}$) (Table 4). Restricting the agent to the meta-template $\mathcal{M}_0$ severely caps quality; expanding $\mathcal{L}$ to five specialized families substantially reduces cross-file inconsistencies. Debug Skill shows a similar progression: post-execution-only fixes reach BH = 69.5, while the Full Living Protocol—which additionally performs lightweight *pre-execution* validations against high-frequency inconsistency classes (e.g., mismatched asset keys)—pushes the system to $72.4/67.2/65.1$. Accumulated experience in both scaffolding and debugging is therefore essential.

Debugging iterations. Figure 3 shows that all metrics improve monotonically with the iteration budget T, with the steepest gains between $T{=}0$ and $T{=}3$ before plateauing at $T{=}5$.

4.5 Qualitative Analysis and Genre Breakdown

Figure 4 decomposes Intent Alignment across five genres. OpenGame is strongest in physics-centric, spatially grounded environments (Platformers 76.8, Top-Down Shooters 71.4), where the specialized template families correctly bind collision layers, physics bodies, and velocity vectors. Both OpenGame and the Cursor baseline degrade noticeably on more abstract genres (Strategy 58.2, Puzzle/UI 52.6; Arcade classics sit at 66.5), where

Table 3: Ablation of the core OpenGame agent workflow mechanisms. Removing structural constraints leads to performance degradation.

Agent Configuration	Build Health	Visual Usability	Intent Alignment
OpenGame (Full Workflow)	**72.4**	**67.2**	**65.1**
w/o Hook-Driven Implementation	62.3	57.6	53.5
w/o Three-Layer Reading	67.8	61.9	56.5
w/o Physics-First Classification	70.2	64.6	61.6

Table 4: Ablation of Agent Evolution (Game Skills). We decompose Template Skill ($\mathcal{L}$) into stages of library maturity, and Debug Skill ($\mathcal{P}$) into its active components.

Template Architecture ($\mathcal{L}$)	Debugging Strategy ($\mathcal{P}$)	Build Health	Visual Usability	Intent Alignment
Static Skeleton ($\mathcal{M}_0$)	Static Rule Checklist	60.5	54.8	51.2
Static Skeleton ($\mathcal{M}_0$)	Full Living Protocol ($\mathcal{P}$)	65.4	59.2	56.3
Partial Evolved Library (2 Families)	Static Rule Checklist	63.1	57.3	53.8
Full Evolved Library (5 Families)	Static Rule Checklist	66.3	60.7	57.9
Full Evolved Library (5 Families)	Post-Execution Fixes Only	69.5	63.8	61.4
Full Evolved Library (5 Families)	**Full Living Protocol ($\mathcal{P}$)**	**72.4**	**67.2**	**65.1**

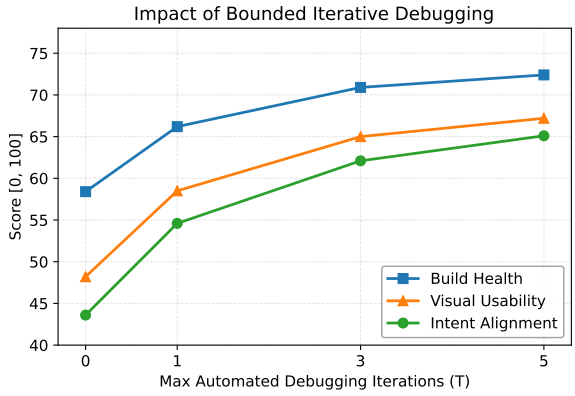

Figure 3: Performance metrics as a function of the maximum allowed automated debugging iterations (T).

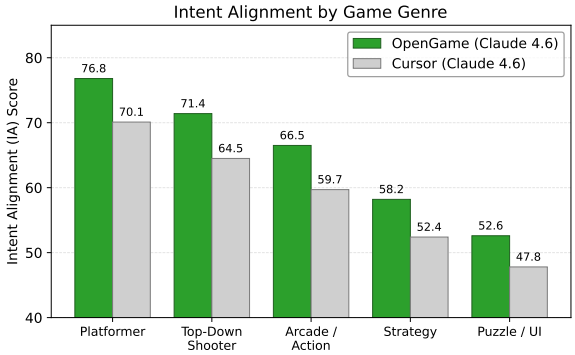

Figure 4: Intent Alignment (IA) scores across different game genres, comparing OpenGame against the Cursor baseline.

logical state—e.g., inventory tracking or match-three rules—is weakly coupled to visible rendering and silent desynchronization failures evade both compiler warnings and runtime crashes, making them hard to detect during automated debugging and highlighting an important direction for future work.

5 Conclusion

In this paper, we present OpenGame, an open-source agentic framework for end-to-end web game creation from natural-language prompts. By combining a structured multi-phase workflow with Game Skill—Template Skill for stable project scaffolding and Debug Skill for cumulative error repair—and a domain-specialized backbone, OpenGame substantially improves the ability of code agents to turn creative design intent into fully playable interactive systems. We further introduce OpenGame-Bench, a dynamic evaluation pipeline that measures build health, visual usability, and intent alignment beyond static code correctness. These results suggest that reliable game generation requires not only stronger code models, but also persistent structural priors, reusable debugging knowledge, and evaluation protocols grounded in real, interactive execution environments.

Limitations

OpenGame currently targets 2D browser games built on the Phaser engine and does not yet extend to 3D titles or proprietary engines such as Unity and Unreal, whose binary asset pipelines remain difficult for text-based agents. Its strongest results require a frontier closed-source backbone (Claude Sonnet 4.6); our domain-specialized GameCoder-27B narrows but does not close this gap, particularly on Intent Alignment. Performance also degrades on abstract, logic-heavy genres such as Strategy and Puzzle/UI, where state desynchronization fails silently and is hard to detect during automated debugging. Finally, our evaluation relies on a VLM judge over 150 prompts across five genres, which can introduce systematic biases and does not fully substitute for human play-testing, and the full multimodal asset-generation and iterative debug loop is substantially more expensive than single-shot generation. Generated assets are not yet checked for originality or copyright, so any downstream deployment should incorporate appropriate safeguards.

References

Anthropic. 2026. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: 2026-04-20.

Anysphere. 2024. Cursor: The ai code editor. https://www.cursor.com. Accessed: 2026-04-20.

Jake Bruce, Michael Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, Chris Apps, Yusuf Aytar, Sarah Bechtle, Feryal Behbahani, Stephanie Chan, Nicolas Heess, Lucy Gonzalez, Simon Osindero, Sherjil Ozair, Scott Reed, and 6 others. 2024. Genie: Generative interactive environments. *Preprint*, arXiv:2402.15391.

Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. 2002. Deep blue. *Artificial Intelligence*, 134(1–2):57–83.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2025. Mle-bench: Evaluating machine learning agents on machine learning engineering. *Preprint*, arXiv:2410.07095.

Wayne Chi, Yixiong Fang, Arnav Yayavaram, Siddharth Yayavaram, Seth Karten, Qiuhong Anna Wei, Runkun Chen, Alexander Wang, Valerie Chen, Ameet Talwalkar, and Chris Donahue. 2026. Gamedevbench: Evaluating agentic capabilities through game development. *Preprint*, arXiv:2602.11103.

Cognition AI. 2024. Devin: The first ai software engineer. https://www.cognition-labs.com/introducing-devin.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3405 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

Richard Davey and Photon Storm. 2013. Phaser - a fast, fun and free open source html5 game framework. https://phaser.io.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. DeepSeek-V3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Epic Games. 1998. Unreal engine. https://www.unrealengine.com.

Mahdi Farrokhi Maleki and Richard Zhao. 2024. Procedural content generation in games: A survey with insights on emerging LLM integration. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 20(1):167–178.

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

Google DeepMind. 2026. Gemini 3.1 Pro: Announcing our latest Gemini AI model. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/. Accessed: 2026-04-20.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Seth Karten, Jake Grigsby, Tersoo Upaa Jr, Junik Bae, Seonghun Hong, Hyunyoung Jeong, Jaeyoon Jung, Kun Kerdthaisong, Gyungbo Kim, Hyeokgi Kim, Yujin Kim, Eunju Kwon, Dongyu Liu, Patrick Mariglia, Sangyeon Park, Benedikt Schink, Xianwei Shi, Anthony Sistilli, Joseph Twin, and 12 others. 2026. The pokeagent challenge: Competitive and long-context learning at scale. *Preprint*, arXiv:2603.15563.

Seth Karten, Andy Luu Nguyen, and Chi Jin. 2025. PokéChamp: an expert-level minimax language agent. *Preprint*, arXiv:2503.04094.

Kimi Team. 2026. Kimi K2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visually grounded web tasks. *Preprint*, arXiv:2401.13649.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *Preprint*, arXiv:2601.11868.

MiniMax. 2026. MiniMax M2.5: Built for real-world productivity. https://www.minimax.io/news/minimax-m25. Accessed: 2026-04-20.

Nunu AI. 2024. Beating the world record in pokémon emerald: An AI agent case study. https://nunu.ai/case-studies/pokemon-emerald.

OpenAI. 2025. GPT-5.1. https://openai.com/index/gpt-5-1/. Accessed: 2026-04-20.

Qwen Team. 2025. Qwen Code: A command-line ai workflow tool for agentic coding. https://github.com/QwenLM/qwen-code. Accessed: 2026-04-20.

Qwen Team. 2026. Qwen3.5-Max-Preview unveiled. https://qwen.ai/blog?id=qwen3.5-max-preview. Accessed: 2026-04-20.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. 2025. Design2Code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3956–3974, Albuquerque, New Mexico. Association for Computational Linguistics.

SIMA Team. 2025. SIMA 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*.

Shyam Sudhakaran, Miguel González-Duque, Matthias Freiberger, Claire Glanois, Elias Najarro, and Sebastian Risi. 2023. Mariogpt: Open-ended text2level generation through large language models. *Preprint*, arXiv:2302.05981.

Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2018. Procedural content generation via machine learning (pcgml). *Preprint*, arXiv:1702.00539.

Unity Technologies. 2005. Unity game engine. https://unity.com.

Alexander Sasha Vezhnevets, John P Agapiou, Avia Aharon, Ron Ziv, Jayd Matyas, Edgar A Duéñez-Guzmán, William A Cunningham, Simon Osindero, Danny Karmon, and Joel Z Leibo. 2023. Generative agent-based modeling with actions grounded in physical, social, or digital space using concordia. *arXiv preprint arXiv:2312.03664*.

Alexander Sasha Vezhnevets, Jayd Matyas, Logan Cross, Davide Paglieri, Minsuk Chang, William A Cunningham, Simon Osindero, William S Isaac, and Joel Z Leibo. 2025. Multi-actor generative artificial intelligence as a game engine. *arXiv preprint arXiv:2507.08892*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. 2024a. Swe-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2024b. Swe-bench multimodal: Do ai systems generalize to visual software domains? *Preprint*, arXiv:2410.03859.

John Yang, Kilian Lieret, Joyce Yang, Carlos E. Jimenez, Muhtasham Oblokulov, Aryan Siddiqui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. CodeClash: Benchmarking goal-oriented software engineering. *Preprint*, arXiv:2511.00839.

Zhipu AI. 2025. GLM-4.5: Advancing Open Bilingual Foundation Models. https://z.ai/blog/glm-4.5. Accessed: 2026-04-20.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. WebArena: A realistic web environment for building autonomous agents. *Preprint*, arXiv:2307.13854.

Hongda Zhu, Yiwen Zhang, Bing Zhao, Jingzhe Ding, Siyao Liu, Tong Liu, Dandan Wang, Yanan Liu, and Zhaojian Li. 2025. Frontendbench: A benchmark for evaluating llms on front-end development via automatic evaluation. *ArXiv*, abs/2506.13832.

Appendix

The appendix is organized as follows. Section A provides the AI assistance disclosure. Section B gives the full Game Skill execution algorithm. Section C documents the construction of the training corpora used by GameCoder-27B, including continual pre-training (CPT), supervised fine-tuning (SFT), and reinforcement learning (RL) data. Section D details the implementation of OpenGame-Bench, including benchmark distribution, the headless evaluation pipeline, and the exact computation of the Build Health, Visual Usability, and Intent Alignment metrics. Section E reproduces the system prompt specifications used by the agent framework, and Section F includes the full prompt files as embedded PDFs.

A AI Assistance Disclosure

In accordance with the ACL policy on the use of AI writing assistants, we disclose that GPT-5.5 was used *solely* for surface-level language polishing of author-written text (e.g., minor grammar and phrasing refinements). No AI assistant was used for ideation, research design, literature search, experimental analysis, code generation, figure or table creation, or for drafting any original technical content. All scientific contributions, results, and conclusions are entirely the work of the human authors, who take full responsibility for the content of this paper.

B Game Skill Execution Algorithm

Algorithm 1 formalizes the Game Skill loop introduced in Section 3.3. Template Skill provides the structural starting point (line 1), and Debug Skill drives the verify–diagnose–repair loop (lines 4–9), with verified fixes appended back to the protocol $\mathcal{P}$ so that debugging knowledge accumulates across tasks.

Algorithm 1: Game Skill execution

Input: User specification x, meta template $\mathcal{M}_0$, template library $\mathcal{L}$, debug protocol $\mathcal{P}$
Output: Runnable game project y

1: Select a template family $T \in \mathcal{L}$ (initialized as $\mathcal{M}_0$ at the beginning of training)
2: Instantiate T to scaffold a project skeleton y
3: Generate game-specific content conditioned on x within the extension points of y
4: **repeat**
5: Run verification and execution (build, test, run) guided by $\mathcal{P}$
6: **if** failure observed **then**
7: Diagnose the failure using $\mathcal{P}$ and repair y
8: Append a verified (signature, cause, fix) entry to $\mathcal{P}$ if the pattern is new
9: **end if**
10: **until** y is buildable and runnable
11: Optionally extract reusable fragments from y and merge into $\mathcal{L}$
12: **return** y

Together, lines 1–3 instantiate a stable, family-specific scaffold from the evolved template library, while lines 4–10 capture the cumulative-debugging behavior that distinguishes Game Skill from a fixed checklist: failure signatures are diagnosed against $\mathcal{P}$, fixes are applied, and any genuinely new (signature, cause, fix) entry is appended back so subsequent tasks benefit. Line 11 optionally feeds reusable fragments back into the template library, closing the agent-evolution loop described in Section 3.3.

C Training Dataset Construction

This section describes the three corpora that drive the three stages of the GameCoder-27B training pipeline introduced in Section 3: the continual pre-training (CPT) corpus, the supervised fine-tuning (SFT) dataset, and the reinforcement learning (RL) environment. All numbers reported below correspond to the version of the data used to train the GameCoder-27B checkpoint evaluated in the main results table.

C.1 Continual Pre-Training (CPT) Corpus

Sources. The CPT corpus is assembled from four complementary sources, each contributing a different slice of game-development knowledge: (i) open-source Phaser 3 repositories on GitHub, (ii) general JavaScript and TypeScript game repositories that do not necessarily use Phaser, (iii) official documentation (Phaser API reference, MDN web docs, and TypeScript handbook), and (iv) community tutorials and blog posts collected from publicly

indexed educational websites.

Cleaning pipeline. We apply a four-stage cleaning pipeline to all collected data: (1) *license filtering*, keeping only permissively licensed material (MIT, Apache-2.0, BSD, and CC-BY variants); (2) *near-duplicate removal* via MinHash with a Jaccard threshold of 0.85 over 8-gram shingles; (3) *syntactic validity*, where every JavaScript and TypeScript file is parsed with the official TypeScript compiler and discarded on parse failure; and (4) *file-level filters* that drop binaries, files larger than 512 KB, files with more than 40% non-ASCII content, and files with fewer than five non-whitespace lines.

Final statistics. Table 5 reports the final size of the CPT corpus after the above cleaning pipeline. Token counts are measured with the Qwen3 tokenizer.

Table 5: Composition of the Continual Pre-Training (CPT) corpus. Token counts are reported in billions (B), measured with the Qwen3 tokenizer.

Source	# Files	Tokens (B)
Phaser 3 repositories	120,000	0.6
JavaScript / TypeScript games	65,000	0.3
Official documentation	5,000	0.1
Community tutorials / blogs	10,000	0.1
Total	**0.2M**	**1.1**

C.2 Supervised Fine-Tuning (SFT) Dataset

Synthesis pipeline. The SFT dataset is synthesized via a two-model distillation pipeline. We use *gpt-codex5.1* as a *prompt-author model* to produce diverse, multi-step game design and implementation prompts (e.g., *"implement a 2D platformer controller with double-jump, coyote-time, and sprite animation tied to velocity"*). For each prompt, we use *minimax2.5* as a *target-author model* to generate a high-quality candidate response. The prompt-author is seeded with archetype tags, asset constraints, and difficulty levels sampled to match the natural-language distribution of OpenGame-Bench so that SFT and evaluation distributions are well-aligned.

Scope. The SFT stage operates exclusively at the *single-file gameplay logic* level (e.g., a controller, a state machine, or a collision handler). We intentionally exclude multi-file project skeletons, asset/config integration, and debug-repair turns from SFT, as these capabilities are obtained downstream either from the agentic framework itself or from the execution-grounded RL stage; restricting SFT to self-contained gameplay components yields a cleaner instruction-following signal and avoids entangling the model with project scaffolding decisions.

Quality filtering. Candidate $(\text{prompt}, \text{response})$ pairs are filtered by a three-stage rubric. (1) *Static check*: the response must contain syntactically valid TypeScript and must reference at least one Phaser core symbol when the prompt is engine-typed. (2) *Build check*: every sample is compiled with a sandboxed `tsc` build under a 60 s budget and discarded on type-check failure. (3) *LLM rubric*: a separate LLM judge scores design fidelity, completeness, and code clarity, and we keep only the top 70% of samples within each archetype.

Archetype distribution. The SFT distribution is deliberately balanced across the five physics archetypes used by the agent's classifier (Section 3.2) to prevent template-family collapse. The final ratios are: *platformer* 28%, *top-down* 22%, *grid logic* 18%, *tower defense* 16%, and *UI-heavy* 16%.

C.3 Reinforcement Learning (RL) Environment

Problem set. The RL stage operates at the level of *single-file gameplay components* rather than whole games. We extract $6{,}500$ component-level problems from the SFT distillation pool, covering five recurring categories: collision and physics body construction, finite-state machines, animation state graphs, spawner / wave logic, and input-action handlers. Each problem ships with a set of 4–12 executable unit tests (mean 7.3) written against a deterministic Phaser test harness.

Reward. For each rollout, the reward is

$$r = \alpha\,\mathbf{1}[\text{build pass}] + \beta \cdot \frac{\#\text{tests passed}}{\#\text{tests total}} - \gamma \cdot \mathbf{1}[\text{timeout}],$$

with $\alpha = 0.3$, $\beta = 0.7$, and a timeout penalty $\gamma = 0.2$ applied when the test process exceeds 20 s. The build flag is required to be true for any positive test-pass credit, which prevents trivial reward hacking by generating code that bypasses compilation.

Training setup. We use a GRPO-style on-policy update with group size 8, KL coefficient 0.05 to the SFT reference policy, and a learning rate of

2×10^{-6}. Because each rollout requires a full build-and-test execution cycle on the deterministic Phaser harness (mean wall-clock $\approx$ 25 s per rollout, peaking at the 20 s test-timeout plus build overhead), the per-step cost of RL is substantially higher than for purely textual code-generation tasks and effectively caps the total number of optimization steps we can afford. In practice we run RL for only 100 optimization steps, generating approximately 800 rollouts in total; the final GameCoder-27B checkpoint is selected by held-out test pass rate on a 500-problem validation split.

D OpenGame-Bench: Implementation Details

This section documents the construction of the OpenGame-Bench benchmark (Section D.1), the headless evaluation pipeline (Section D.2), and the exact computation of the three metrics (Section D.3) reported throughout the main paper.

D.1 Benchmark Construction and Distribution

Sourcing. OpenGame-Bench consists of 150 natural-language game design prompts. The vast majority (130 prompts) are *human-crafted* from scratch by the authors and a small group of in-house annotators with prior game-design experience. Only a small fraction (20 prompts) are adapted from publicly available game-jam repositories on `itch.io` and GitHub, where we lift the original human-written "concept" or "pitch" sections and paraphrase them into self-contained design specifications stripped of any implementation hints; no prompt is generated by an LLM. All prompts are written or rewritten under an authoring rubric requiring (i) a high-level theme statement, (ii) one or more compulsory mechanics, and (iii) optional stylistic notes, and are then cross-reviewed by a second annotator for clarity, technical feasibility under a 2D web stack, and the absence of intellectual-property references. Authoring deliberately avoids any code, file names, or framework hints so that the resulting specifications remain purely design-level inputs.

Genre coverage. Table 6 reports the per-genre distribution. Genres are intentionally broader than the agent's five *physics archetypes* (Section 3.2) so that the benchmark does not reward systems for memorizing exactly the same taxonomy.

Table 6: OpenGame-Bench prompt distribution by genre and difficulty. Difficulty is the number of compulsory mechanics extracted from each prompt during the IA spec compilation step (Section D.3).

Genre	#	Easy	Med.	Hard
Platformer	35	10	18	7
Top-down shooter	30	8	16	6
Puzzle / UI	28	10	14	4
Arcade classics	30	12	13	5
Strategy	27	2	17	8
Total	**150**	42	78	30

Prompt statistics. The average prompt is 128 words ($\sigma = 41$); the shortest is 52 words and the longest is 284. Each prompt specifies (i) a high-level theme, (ii) one or more compulsory mechanics, and (iii) optional stylistic notes; no prompt contains code, file names, or framework hints.

D.2 Evaluation Pipeline Implementation

Execution stack. Each generated project directory is evaluated inside a hermetic Docker container based on `node:20-bookworm`. The container's entrypoint performs the following steps:

1. **Install & build.** If a `package.json` is present we run `npm install` (with a frozen lockfile when available) followed by `npm run build`; otherwise the project is assumed to be a static `index.html` bundle.
2. **Serve.** A local `http-server` is started on `127.0.0.1:8080` rooted at the build output directory.
3. **Launch.** A headless Chromium instance is launched via `Playwright` at viewport 1280×800 with a fixed user agent. The page must reach the `networkidle` state within 30 s.
4. **Play.** During a 60 s interaction window, a deterministic fuzzer injects keyboard and mouse events sampled from a small distribution of common control schemes (WASD, arrow keys, space, mouse-left, and mouse-move).
5. **Capture.** Six full-page screenshots are taken at fixed intervals (every 10 s starting at $t = 5$ s), and the browser `console` and `network` event streams are logged.

Stochastic averaging. To account for stochasticity in both generation and the fuzzer, every task is evaluated three times with different random seeds. Reported metrics are means across the three suc-

cessful runs; if more than one run fails the precondition checks the task is recorded as a pipeline error rather than a zero.

D.3 Metric Computation

All three metrics are scaled to $[0, 100]$.

Build Health (BH). BH is a weighted combination of static signals captured during the install/build/launch phase:

$$\mathrm{BH} = 100 \cdot \big(w_b \cdot s_b + w_r \cdot s_r + w_n \cdot s_n + w_v \cdot s_v\big),$$

where $s_b \in \{0, 1\}$ indicates that the build succeeded, s_r is $1 - \min(1, k_r/K_r)$ with k_r the number of unhandled JavaScript runtime exceptions and $K_r = 5$ a saturation constant, s_n is the analogous network-error term with $K_n = 3$, and $s_v \in \{0, 1\}$ indicates that the <canvas> produced at least one non-empty frame. The weights are $(w_b, w_r, w_n, w_v) = (0.4, 0.25, 0.15, 0.20)$, calibrated on a 30-task pilot so that "builds but renders nothing" and "renders but spams runtime errors" both score in the low-50s.

Visual Usability (VU). VU combines a pixel-level heuristic with a Vision-Language Model (VLM) judge:

$$\mathrm{VU} = 100 \cdot \big(w_e \cdot \bar{H} + w_m \cdot \bar{M} + w_j \cdot \bar{V}\big),$$

where $\bar{H} \in [0, 1]$ is the mean per-channel Shannon entropy of the six captured frames (normalized by $\log_2 256$ and clipped), $\bar{M} \in [0, 1]$ is the mean fraction of pixels whose RGB distance between consecutive frames exceeds a threshold $\tau = 12$, and $\bar{V} \in [0, 1]$ is the mean of a VLM judge score (GPT-5-mini, 0–5, normalized) prompted to rate visual coherence, animation quality, and apparent interactability. We use weights $(w_e, w_m, w_j) = (0.20, 0.20, 0.60)$.

Intent Alignment (IA). IA is computed in two steps. First, a *requirement compiler* (an LLM prompted with a structured rubric, then validated by a deterministic JSON schema) converts each prompt into a list of n atomic mechanical requirements $\{r_1, \ldots, r_n\}$, each annotated with an importance weight $w_i \in \{1, 2, 3\}$ (cosmetic / core / critical). Second, for each r_i a *VLM verifier* (GPT-5-mini) is shown the six captured screenshots and the relevant console log slice, and returns a verdict $v_i \in \{\text{pass}, \text{partial}, \text{fail}\}$, scored as 1, 0.5, 0 respectively. The final score is the importance-weighted pass rate

$$\mathrm{IA} = 100 \cdot \frac{\sum_{i=1}^{n} w_i \cdot v_i}{\sum_{i=1}^{n} w_i}.$$

The requirement compiler is run once per prompt and cached; the VLM verifier is rerun for every evaluated system to prevent leakage.

Implementation. The full pipeline is implemented in Python with a single `evaluate(project_dir, prompt) -> Dict[str, float]` entrypoint. The average wall-clock cost is $\approx$ 220 s per task per seed, dominated by the 60 s interaction window and the VLM judging calls. All judge prompts, requirement-compiler prompts, and Docker manifests will be released alongside the benchmark.

E System Prompt Specifications

This section presents the prompt specifications used in the OpenGame agent framework, reproduced from the source files used during evaluation.

E.1 Main System Prompt

The main system prompt is injected as the agent's system-level instruction via `agent-test/custom.md`. It defines the complete autonomous six-phase workflow for 2D game development:

1. **Classification & Scaffolding** — invoke `classify-game-type` and copy the corresponding template family into the workspace.
2. **Game Design** — generate a technical GDD via `generate-gdd`, then expand per-file todos from GDD Section 5.
3. **Asset Synthesis** — call `generate-game-assets` and `generate-tilemap` based on the GDD asset registry and ASCII maps.
4. **Config & Registration** — merge `gameConfig.json` and register all scenes in `main.ts` / `LevelManager.ts`.
5. **Code Implementation** — three-layer reading strategy (API summary → targeted source → implementation guide), followed by hook-based coding against template files.
6. **Verification** — static self-review checklist from `debug_protocol.md`, then `npm run build`, `npm run test`, and `npm run dev`.

E.2 Game Classification Tool Prompt

This tool classifies a user's game idea into one of five archetypes using *Physics-First Logic* (gravity, perspective, and movement type) rather than genre names. It calls an external LLM (DeepSeek-v3.2 by default) and returns a structured JSON result. The compiled PDF contains three prompts in order:

1. **Tool Description** — the one-line capability summary and parameter list exposed to the agent as a tool manifest entry.
2. **System Prompt** — classification rules for five archetypes (*platformer*, *top_down*, *grid_logic*, *tower_defense*, *ui_heavy*), each with a key discriminating question, physics profile, and common-mistake warnings.
3. **User Prompt** — the runtime template that wraps the user's game description and requests a JSON-only response.

E.3 GDD Generation Tool Prompt

This tool generates a technical game Design Document (GDD) tailored to a specific archetype. The system prompt is dynamically assembled from a fixed header plus three documents loaded from disk: `docs/gdd/core.md` (universal 6-section GDD format), `docs/modules/{archetype}/design_rules.md` (game design guide), and `docs/modules/{archetype}/template_api.md` (code capability list). The compiled PDF contains eight prompts in order:

1. **Tool Description** — function signature and required parameters (`raw_user_requirement`, `archetype`).
2. **System Prompt – Fixed Header** — instructs the model to act as a game design engineer and enforces four core rules: user-faithful, config-first, zero custom code, and hook integrity.
3. **User Prompt** — runtime template requesting a 6-section Technical GDD with archetype-specific guidance injected at call time.
4. **Section 1 Asset Guidance – Platformer** — side-view animation frames, tileset grid format, and audio SFX list.
5. **Section 1 Asset Guidance – UI Heavy** — front-view bust shots, per-expression image naming, and UI audio conventions.
6. **Section 1 Asset Guidance – Top-Down** — directional animation triplets and tilemap-vs-arena sub-mode rules.
7. **Section 1 Asset Guidance – Grid Logic** — strict `type:"image"` parameter constraints and background overlay model.
8. **Section 1 Asset Guidance – Tower Defense** — tower, enemy, projectile, and icon asset conventions with correct JSON examples.

E.4 Todo List Tool Prompt

This tool creates and manages a structured task list for the agent's coding session, enabling real-time progress tracking across multi-phase workflows. Parameters: `todos` array of items with `id`, `content`, and `status` (`pending`/`in_progress`/`completed`). The compiled PDF contains two prompts in order:

1. **Tool Description** — the capability summary exposed to the agent as a tool manifest entry.
2. **Full Tool Prompt** — comprehensive guidance on when to use the todo list (3+ step tasks, multi-file refactors, game development pipelines), worked examples of both correct and incorrect usage, and task state management rules (one in-progress at a time; mark complete immediately upon finishing).

E.5 Asset Generation Tool Prompts

This tool generates game assets (images, animations, audio, tilesets, backgrounds) using AI vision and audio models (Tongyi / Doubao backends). Features include automatic background removal, image-to-video (I2V) animation generation, and ABC-notation-based music synthesis. The compiled PDF contains seven prompts, one per asset type, in order:

1. **Tool Description** — supported asset types, model backends, and key pipeline features.
2. **Background Generation** — full-scene, edge-to-edge illustration prompt; explicitly forbids characters, UI elements, and transparency.
3. **Image (Sprite) Generation** — single isolated object on a pure white background with centered composition.
4. **Animation Base Image** — side-view chibi character in neutral idle pose; used as the seed frame for the I2V pipeline.
5. **Animation Frame – I2V (Image-to-Video)** — motion description for the image-to-video model; enforces consistent side-view framing and identical character size across frames.
6. **Animation Frame – I2I (Image-to-Image)** —

per-frame prompt with frame index and total count for the image-to-image pipeline.

7. **Tileset Generation** — 3×3 seamless tileset with strict row/column layout, zero gaps, full 1024×1024 canvas coverage, and forbidden elements list.

E.6 Audio Generation Prompts (ABC Notation)

The audio generation pipeline uses a two-step process: (1) generate ABC music notation via LLM, then (2) convert the ABC notation to WAV using symusic/Python. The compiled PDF contains two prompts in order:

1. **ABC System Prompt** — mandatory header fields (`X:`, `T:`, `M:`, `L:`, `Q:`, `K:`), note-length and rest syntax reference, and a valid two-part example; instructs the model to produce loop-friendly game music with actual note sequences (not placeholders).
2. **ABC Generation Prompt (User Message)** — runtime template specifying duration, audio type (BGM/SFX), genre, tempo, and description; requests a JSON response with `notation` and `comments` fields and provides good/bad notation examples.

E.7 Tilemap Generation Tool

This is a *purely algorithmic* tool with no LLM prompts. It converts ASCII map layouts into Phaser Tilemap JSON files using 47-tile blob auto-tiling (bitmasking). Key parameters: `tileset_key`, `tile_size` (default 64), `tileset_grid_size` (default 7), `auto_tiling`, `auto_tile_chars` (default `["#"]`), `mode` (`"floor"` or `"walls"`), and a `maps` array of map definitions with `map_key`, `layout_ascii`, `legend`, and `object_markers`. The compiled PDF contains one item:

1. **Tool Description** — the capability summary exposed to the agent as a tool manifest entry.

E.8 GDD Built-in Archetype Rules (Fallback)

When the external design rule documents (`design_rules.md`, `template_api.md`) are not found on disk, the GDD generator falls back to these built-in rules. All five archetypes are embedded verbatim in the source code; due to length, only the platformer rules are reproduced below. The compiled PDF contains one item:

1. **Platformer Rules (Built-in)** — physics settings (Y-axis gravity, side view), available behaviors (`PlatformerMovement`, `MeleeAttack`, `RangedAttack`, `PatrolAI`, `ChaseAI`), nine ultimate skill types, ASCII level design legend with placement constraints, and the canonical `gameConfig.json` schema using the `{ "value": X }` wrapper format.

F Prompt Appendix Pages

```
## Platformer Rules (Built-in)

### Physics
- Gravity: ON (Y-axis), side view, Left/Right + Jump

### Available Behaviors
- PlatformerMovement (walkSpeed, jumpPower; optional: doubleJumpEnabled, coyoteTime, jumpBufferTime)
- MeleeAttack (damage, range, cooldown)
- RangedAttack (damage, projectileKey, projectileSpeed)
- PatrolAI, ChaseAI

### Ultimate Skills (pick one per character)
DashAttackSkill, AreaDamageSkill, TargetedAOESkill, TargetedExecutionSkill, BeamAttackSkill,
 ↪ GroundQuakeSkill, BoomerangSkill, MultishotSkill, ArcProjectileSkill. All share base config: id,
 ↪ name, cooldown.

### Level Design (ASCII)
- Legend: # = Ground, = = Platform, . = Air, P = Player, E = Enemy, B = Boss, C = Coin
- Entities FALL. Don't place enemies in mid-air.
- MUST copy from predefined templates (A/B/C/D), NEVER invent from scratch

### Config Schema
```json
{
  "playerConfig": {
    "maxHealth": { "value": 100 }, "walkSpeed": { "value": 360 },
    "jumpPower": { "value": 2400 }, "attackDamage": { "value": 40 },
    "gravityY": { "value": 1200 }, "hurtingDuration": { "value": 100 },
    "invulnerableTime": { "value": 1000 }
  },
  "enemyConfig": { "maxHealth": { "value": 50 }, "walkSpeed": { "value": 80 }, "damage": { "value":
   ↪ 20 } },
  "bossConfig": { "maxHealth": { "value": 200 }, "damage": { "value": 35 }, "speed": { "value": 120 }
   ↪ },
  "ultimateConfig": { "cooldown": { "value": 5000 }, "damage": { "value": 80 } }
}
```
All values use `{ "value": X, "type": "...", "description": "..." }` wrapper format.

Hook Integrity
Every hook name MUST exist in template_api.md. Do NOT invent hooks.
```

```
You are a game coding agent specializing in 2D game development tasks. Your primary goal is to help
 ↪ users safely and efficiently, adhering strictly to the following instructions and utilizing your
 ↪ available tools.

# 2D Game Development: CODE-FIRST MODE

**When creating a 2D game, you MUST work autonomously until completion.**

**Key Principle**: Template architecture informs GDD design. Full template code is read only at
 ↪ implementation time.

---

## WORKFLOW (Execute in Order)

**First action**: Use `todo_write` to plan your full workflow, then execute each phase below. Update
 ↪ todos as you progress.

### Phase 1: Classification and Scaffolding

1. **Classify**: Call `classify-game-type` tool with user's game idea.

Uses **Physics-First Logic** (not genre names):

| Module | Physics | Key Question | Examples |
|--------|---------|--------------|----------|
| `platformer` | Side + Gravity | Does character FALL? | Mario, Terraria, Street Fighter |
| `top_down` | Top-Down + Free | Can move UP without jump? | Zelda, Isaac, Vampire Survivors |
| `grid_logic` | Grid + Discrete | Snap to grid? | Sokoban, Fire Emblem, Match-3 |
| `tower_defense` | Path + Waves | Fixed enemy paths? | Kingdom Rush, Bloons TD |
| `ui_heavy` | UI / No Physics | Primarily UI? | Card games, Visual Novels |

2. **Scaffold**: Use `run_shell_command` to copy templates and docs (FOUR steps, in order):

```bash
# Step 1: Copy core template (creates src/, public/, config files)
cp -r {TEMPLATES_DIR}/core/* ./

# Step 2: Copy module-specific code INTO src/ (ADDITIVE merge)
cp -r {TEMPLATES_DIR}/modules/{archetype}/src/* ./src/

# Step 3: Copy core documentation
mkdir -p docs/gdd
cp {DOCS_DIR}/gdd/core.md docs/gdd/
cp {DOCS_DIR}/asset_protocol.md {DOCS_DIR}/debug_protocol.md docs/

# Step 4: Copy module-specific documentation
mkdir -p docs/modules/{archetype}
cp -r {DOCS_DIR}/modules/{archetype}/* docs/modules/{archetype}/
```

- Do NOT manually create directories -- they come from templates automatically
- **After copying, proceed DIRECTLY to Phase 2. Do NOT read any source files yet** -- template code
 ↪ is only read in Phase 5. Reading now wastes context window.

Phase 2: Game Design

3. **Call `generate-gdd`** with:
   - `raw_user_requirement`: User's game idea
   - `archetype`: From Phase 1 classification (REQUIRED)

The tool auto-loads three documents:
- `{DOCS_DIR}/gdd/core.md` -- universal GDD format
- `{DOCS_DIR}/modules/{archetype}/design_rules.md` -- game design guide (gameplay, flow, feel)
- `{DOCS_DIR}/modules/{archetype}/template_api.md` -- code capability list (systems, hooks,
 ↪ components)
(Tool searches for `docs/` folder from project root upward)

4. **Save GDD** to `GAME_DESIGN.md` using `write_file` tool

5. **Expand todos NOW**: GDD exists -- replace the IMPLEMENT placeholder with **specific per-file
 ↪ todos** from GDD Section 5 (each todo = `COPY`/`UPDATE`/`CREATE`/`MERGE` + GDD section
 ↪ reference). Ensure READ and VERIFY phases are still present.

The GDD has 6 sections. Each section feeds a specific downstream step:
- **Section 0** (Architecture) -> Phase 4 scene registration (`main.ts` + `LevelManager.ts`)
- **Section 1** (Assets) -> Phase 3 asset generation
- **Section 2** (Config) -> Phase 4 gameConfig.json
- **Section 3** (Entities/Scenes) -> Phase 5 code implementation
```

```
Classifies a game idea into an archetype based on PHYSICS and PERSPECTIVE (not genre name). Returns:
↪ platformer (side+gravity), top_down (free movement), grid_logic (discrete tiles), tower_defense
↪ (path+waves), or ui_heavy (no physics). Call this FIRST before scaffolding templates.
```

```
# Game Type Classifier (Physics-First Logic)

You are a game physics analyzer. Your job is to classify games based on their PHYSICS and
↪ PERSPECTIVE, not their genre name.

## Classification Rules

### 1. platformer (Side View + Gravity)
**Physics**: Y-axis gravity enabled, characters fall down
**Perspective**: Side view (camera looks from the side)
**Movement**: Left/right + jump
**Examples**: Mario, Angry Birds, Street Fighter, Terraria, Hill Climb Racing, Metal Slug, Flappy Bird

**Key Question**: Does the character FALL if there's no ground beneath them?

### 2. top_down (Top-Down + Free Movement)
**Physics**: No gravity (or negligible), free 8-direction movement
**Perspective**: Top-down or isometric (camera looks from above)
**Movement**: WASD in any direction
**Examples**: Zelda, Binding of Isaac, Vampire Survivors, Asteroids, GTA 2D, Hotline Miami

**Key Question**: Can the character move UP without jumping?

### 3. grid_logic (Grid + Turn/Static Logic)
**Physics**: Minimal physics, snap-to-grid movement
**Perspective**: Usually top-down, but grid-locked
**Movement**: Discrete steps (one tile at a time)
**Examples**: Sokoban, Fire Emblem, Chess, Tetris, Match-3, Minesweeper, Snake

**Key Question**: Does movement happen in discrete grid steps?

### 4. tower_defense (Path + Waves)
**Physics**: Enemies follow predefined paths
**Perspective**: Usually top-down
**Movement**: Path-following for enemies, point-and-click for towers
**Examples**: Kingdom Rush, Bloons TD, Plants vs Zombies

**Key Question**: Do enemies follow a fixed path while player places defenses?

### 5. ui_heavy (UI Driven / No Physics)
**Physics**: Almost no arcade physics
**Perspective**: N/A (UI panels)
**Movement**: Click/tap interactions
**Examples**: Card games (Slay the Spire), Visual Novels, Idle/Clicker games, Rhythm games (note
↪ highways)

**Key Question**: Is the game primarily UI panels and state changes?

## Output Format

Respond with ONLY a JSON object (no markdown, no explanation outside JSON):

{
  "archetype": "platformer" | "top_down" | "grid_logic" | "tower_defense" | "ui_heavy",
  "reasoning": "Brief explanation of why this archetype was chosen based on physics",
  "physicsProfile": {
    "hasGravity": true | false,
    "perspective": "side" | "top_down" | "none",
    "movementType": "continuous" | "grid" | "path" | "ui_only"
  }
}

## Common Mistakes to Avoid

- Terraria is NOT top_down (it has gravity, it's platformer)
- Angry Birds is NOT puzzle (it has gravity physics, it's platformer)
- Hill Climb Racing is NOT top_down (it has gravity, it's platformer)
- SimCity/Factorio are grid_logic (grid-based building), not top_down
- Racing games: If side-view with gravity = platformer, if top-down = top_down
```

```
Generates a Game Design Document (GDD) tailored to a specific game archetype. Call this AFTER
↪ classify-game-type and scaffold. The tool dynamically loads archetype-specific design rules.
```

```
# Game Design Document Generator

You are a game design engineer. Produce a **Technical Game Design Document** -- a specification where
↪ every section maps to a downstream tool input or code file.

**Archetype**: {archetype}

**Core Rules:**
1. **User-Faithful**: Fulfill stated requirements. Do not invent unasked features.
2. **Config-First**: Numeric values go in `gameConfig.json` using `{ "value": X }` wrapper.
3. **Zero Custom Code**: Use existing behaviors/hooks from template_api.md only.
4. **Hook Integrity**: Every hook name MUST exist in template_api.md. Non-existent hooks cause
↪ compilation failure.
```

```
**User's Game Idea**: {raw_user_requirement}

**Archetype**: {archetype}

Generate a Technical GDD with **6 sections** (Section 0-5) following the structure defined in the GDD
↪ Core Rules above. Be specific -- the coding agent should never need to guess.

**Section-specific guidance:**

### Section 1 -- Asset Registry
{archetype-specific asset rules}

### Section 2 -- Game Configuration
MERGE into existing `src/gameConfig.json`. Write only game-specific values -- do NOT include
↪ screenSize, debugConfig, renderConfig.

### Section 3 -- {archetype-specific title}
{archetype-specific entity/scene guidance}

### Section 4 -- {archetype-specific title}
{archetype-specific level/content guidance}
```

```
**Platformer asset rules:**
- Characters: `type: "animation"`, SIDE VIEW facing RIGHT, **ONE character per image**, frames:
↪ `idle(2), run(2), jump(2), attack_1(2), attack_2(2), die(2)`
- Backgrounds: `type: "background"`, one per level, `resolution: "1536*1024"`. Keep clean -- no
↪ characters or platforms
- Tilesets: `type: "tileset"`, `tileset_size: 3` (3x3 grid). Keep simple -- complex bricks clutter
↪ the scene
- Projectiles/effects: `type: "image"` for skill projectiles (hammer, missile, boulder) or ranged
↪ bullets
- Audio: BGM per level + SFX for jump, attack, damage, death, coin, victory, ultimate
```

```
**UI Heavy asset rules:**
- Characters: `type: "image"` (NOT animation!), FRONT VIEW bust shot, one image PER expression
- Naming: `{character}_{expression}.png` (e.g., hero_neutral, hero_angry)
- Backgrounds: `type: "background"`, one per scene, `resolution: "1536*1024"`
- Audio: include BGM per scene type + SFX for click, correct, wrong, damage, victory
```

```
**Top-Down asset rules** (see Design Guide Sections 4-5 for full art direction):
- Characters: `type: "animation"`, TOP-DOWN view, **ONE character per image**, 3 directional images
↪ per action (`{name}_{action}_front/back/side`), each with `frameCount: 1`
- **TILEMAP sub-mode**: TWO `type: "tileset"` per theme (`{theme}_floor` + `{theme}_walls`, both
↪ `tileset_size: 3`), plus 2-3 obstacle `type: "image"` + 1-2 decoration `type: "image"`
- **ARENA sub-mode**: NO tilesets. 1 `type: "background"` (`resolution: "1536*1024"`, seamlessly
↪ tileable for scrolling)
- Audio: BGM per level + SFX for combat actions
```

```
**Grid Logic asset rules:**
- **Background**: `type: "background"`, one per level, `resolution: "1536*1024"`. Top-down view of
↪ the environment (dungeon floor, garden, abstract board). The code renders a grid overlay on top
↪ of this image. Description should depict the ENTIRE map surface (e.g., "dark stone dungeon floor
↪ with subtle cracks, top-down view").
```

```
Creates and manages a structured task list for your current coding session. This helps track
 ↪ progress, organize complex tasks, and demonstrate thoroughness.
```

```
Use this tool to create and manage a structured task list for your current coding session. This helps
 ↪ you track progress, organize complex tasks, and demonstrate thoroughness to the user.
It also helps the user understand the progress of the task and overall progress of their requests.

## When to Use This Tool
Use this tool proactively in these scenarios:

1. Complex multi-step tasks - When a task requires 3 or more distinct steps or actions
2. Non-trivial and complex tasks - Tasks that require careful planning or multiple operations
3. User explicitly requests todo list - When the user directly asks you to use the todo list
4. User provides multiple tasks - When users provide a list of things to be done (numbered or
 ↪ comma-separated)
5. After receiving new instructions - Immediately capture user requirements as todos
6. When you start working on a task - Mark it as in_progress BEFORE beginning work. Ideally you
 ↪ should only have one todo as in_progress at a time
7. After completing a task - Mark it as completed and add any new follow-up tasks discovered during
 ↪ implementation

## When NOT to Use This Tool

Skip using this tool when:
1. There is only a single, straightforward task
2. The task is trivial and tracking it provides no organizational benefit
3. The task can be completed in less than 3 trivial steps
4. The task is purely conversational or informational

NOTE that you should not use this tool if there is only one trivial task to do. In this case you are
 ↪ better off just doing the task directly.

## Examples of When to Use the Todo List

<example>
User: I want to add a dark mode toggle to the application settings. Make sure you run the tests and
 ↪ build when you're done!
Assistant: I'll help add a dark mode toggle to your application settings. Let me create a todo list
 ↪ to track this implementation.
*Creates todo list with the following items:*
1. Create dark mode toggle component in Settings page
2. Add dark mode state management (context/store)
3. Implement CSS-in-JS styles for dark theme
4. Update existing components to support theme switching
5. Run tests and build process, addressing any failures or errors that occur
*Begins working on the first task*

<reasoning>
The assistant used the todo list because:
1. Adding dark mode is a multi-step feature requiring UI, state management, and styling changes
2. The user explicitly requested tests and build be run afterward
3. The assistant inferred that tests and build need to pass by adding "Ensure tests and build
 ↪ succeed" as the final task
</reasoning>
</example>

<example>
User: Help me rename the function getCwd to getCurrentWorkingDirectory across my project
Assistant: Let me first search through your codebase to find all occurrences of 'getCwd'.
*Uses grep or search tools to locate all instances of getCwd in the codebase*
Assistant: I've found 15 instances of 'getCwd' across 8 different files. Let me create a todo list to
 ↪ track these changes.
*Creates todo list with specific items for each file that needs updating*

<reasoning>
The assistant used the todo list because:
1. First, the assistant searched to understand the scope of the task
2. Upon finding multiple occurrences across different files, it determined this was a complex task
 ↪ with multiple steps
3. The todo list helps ensure every instance is tracked and updated systematically
4. This approach prevents missing any occurrences and maintains code consistency
</reasoning>
</example>


<example>
User: Create a 2D platformer game about a ninja.
Assistant: I'll create your ninja platformer! Let me plan the full workflow.
```

```
Generates game assets (images, animations, audio) using AI models.
Supports Tongyi and Doubao models. Features auto background removal,
I2V animation generation, and ABC-based music generation.
```

```
Game background art, {description}, {style_anchor}.
Full scene illustration, edge-to-edge, high detail, seamless composition.
IMPORTANT: This is a BACKGROUND - must be fully opaque with rich colors.
Fill entire canvas with scenery.

---FORBIDDEN (CRITICAL)---
NO characters, NO people, NO figures, NO creatures, NO animals, NO NPCs.
NO text, NO UI elements, NO transparency.
PURE SCENERY ONLY - landscape, buildings, environment, sky, etc.
```

```
Game asset sprite, {description}, {style_anchor}.
Single object only, isolated on white background, centered composition, consistent size.
{composition_env}
IMPORTANT: Pure white background, no text, no position offset, ONE object only.
```

```
Game character sprite, {description}, {style_anchor}.
SIDE VIEW (profile view, facing right), chibi style with big head.
Single character only, neutral idle pose, centered composition.
{composition_env}
IMPORTANT: Pure white background, SIDE VIEW only, ONE character only.
```

```
{description}, {action_desc}.
SIDE VIEW animation, single character only, smooth loop.
IMPORTANT: Keep SIDE VIEW, character size MUST remain identical, stay at same position!
{style_anchor}
```

```
Same exact character as reference, {action_desc}.
SIDE VIEW animation, single character only, frame {i} of {frameCount}.
IMPORTANT: Keep SIDE VIEW, same direction as reference.
IMPORTANT: Character size MUST be identical, stay at same position!
{style_anchor}, white background.
```

```
3x3 Seamless Tileset (9 tiles total), {style_anchor}.

THEME: {description}. Use THIS EXACT material/theme for ALL tiles.

#1 SEAMLESS: All 9 tiles MUST touch directly. ZERO gaps. ZERO padding. ZERO spacing.
#2 FULL COVERAGE: Fill ENTIRE 1024x1024 canvas. NO margins. NO white space. NO empty areas.
#3 THEME CONSISTENCY: ALL 9 tiles must match the theme "{description}".

LAYOUT - exactly 3 rows x 3 columns:
Row 1: [TL corner][Top edge][TR corner]
Row 2: [Left edge][Center][Right edge]
Row 3: [BL corner][Bottom edge][BR corner]

GRID CONTENTS (Row by Row):
- Top Row (1,2,3): Surface layer / Top Edges.
- Middle Row (4,5,6): Main Body / Center Fill. MUST BE TILEABLE.
- Bottom Row (7,8,9): Bottom Edges / Hanging base.

STYLE: Flat 2D front view. {style_anchor}. Consistent lighting.

---FORBIDDEN---
text, labels, numbers, letters, watermarks,
grid lines, borders, frames, padding, margins, gaps,
isometric, 3D, perspective, vignette, dark corners.
```

```
You are a helpful assistant that generates creative and original music in ABC notation format.

CRITICAL ABC FORMAT RULES:
1. MUST start with X: (reference number)
2. MUST have T: (title)
3. MUST have M: (meter like 4/4)
4. MUST have L: (default note length like 1/8)
5. MUST have Q: (tempo like 1/4=120)
6. MUST have K: (key like C, G, Am)
7. MUST have actual notes in the body (CDEFGAB or cdefgab)

Notes Reference:
- C D E F G A B = lower octave
- c d e f g a b = higher octave
- c' d' = even higher octave
- C, D, = lower octave
- z = rest
- A2 = double length, A/2 = half length
- | = bar line, |] = end

VALID EXAMPLE:
X:1
T:Game Theme
M:4/4
L:1/8
Q:1/4=120
K:C
|: CDEF GABc | cBAG FEDC | EFGA Bcde | dcBA GFED :|
|: cdef gabc' | c'bag fedc | efga bc'ed | cBAG FEDC :|

Generate catchy, loop-friendly game music with ACTUAL NOTES.
```

```
Generate ABC notation for game audio:

Duration: ~{duration} seconds
Type: {audioType}
Genre: {genre}
Tempo: {tempo}
Description: {description}

Return JSON with:
- notation: Complete ABC notation string
- comments: Brief notes

CRITICAL REQUIREMENTS:
1. notation MUST contain actual notes (CDEFGAB or cdefgab)
2. notation MUST have all required headers (X:, T:, M:, L:, Q:, K:)
3. For BGM: Use repeats |: :| for looping, at least 4-8 bars
4. For SFX: Short melody, 1-2 bars

Example good notation for BGM:
"X:1\nT:Adventure\nM:4/4\nL:1/8\nQ:1/4=120\nK:G\n|: GABc d2BA | GABc d2dc | BAGF E2FG | A4 G4 :|"

Example good notation for SFX:
"X:1\nT:Jump\nM:4/4\nL:1/16\nQ:1/4=180\nK:C\nCEGc c'2z2 |]"
```

```
Converts ASCII layouts into Phaser Tilemap JSON with 47-tile blob autotiling (bitmasking).
```