



JAMER: Project-Level Code Framework Dataset and Benchmark on Professional Game Engines

Jianwen Sun^{1,2}, Chuanhao Li⁴, Zizhen Li^{1,2}, Yukang Feng^{1,2}, Fanrui Zhang², Yifei Huang³, Yu Dai¹, Kaipeng Zhang^{2,3} 

¹Nankai University, Tianjin, China ²Shanghai Innovation Institute, Shanghai, China ³Alaya Studio ⁴Shanghai AI Laboratory, Shanghai, China

Current AI-driven game development has made substantial progress in asset generation, gameplay design, and web-based game coding, yet project-level code engineering on professional game engines remains largely unexplored due to the absence of large-scale datasets and deterministic evaluation methods. We present JamSet and JamBench, the first project-level game code framework dataset and benchmark built on a professional game engine. Our key insight is that Game Jam competitions, community events where developers build complete games under tight time constraints, yield thousands of open-source projects suitable for this purpose. Building on the Godot engine’s text-based format and headless execution mode, we design a deterministic verification pipeline from file integrity to runtime behavior collection, distilling 8,133 verified projects from over 240,000 repositories. Of these, 300 manually verified projects form JamBench; the rest constitute JamSet. JamBench defines theme-driven generation and code completion tasks, evaluated through a pipeline combining compilation pass rates, Structural Completeness Score (SCS), and Behavioral Alignment Score (BAS). Evaluation of 9 frontier models reveals a capability cliff as project scale increases, with runtime pass rates dropping from 80.4% on small projects to 5.7% on large ones (Task2a). Code Agents improve compilation rates yet yield no gains in runtime behavioral quality, indicating that the bottleneck lies in architectural design rather than syntactic correctness. Experiments validate JamSet as effective training data. All data and code are publicly available.

Correspondence: sunjianwen@mail.nankai.edu.cn, kaipeng.zhang@shanda.com

Date: June 19, 2026



1 Introduction

Game development spans multiple interconnected concerns, from art assets and gameplay rules to underlying code frameworks. AI-driven approaches have made substantial progress on assets and gameplay: generative models produce high-quality textures and sprites [7, 13, 28], while procedural content generation [11, 34] and LLM-based rule design [33, 41] continue to advance. On the code side, recent work has explored lightweight web frameworks that generate 2D games in JavaScript or TypeScript [17, 45], and a few efforts target local modifications on professional engines [5]. However, **project-level code framework generation on professional game engines remains largely unexplored**, due to the absence of both large-scale datasets and deterministic evaluation methods.

Two interrelated challenges underlie this gap. The first is evaluation. In traditional software engineering, correctness can be verified through unit testing

against expected outputs [3, 1]. However, game behavior is context-dependent at the project level, with no simple input-output correspondence [6]. Existing alternatives each fall short: hand-written test scripts are precise but prohibitively expensive to scale [5]; VLM- or LLM-based scoring offers scalability but introduces subjectivity and lacks reproducibility [17, 45]; and standard unit testing cannot capture game-specific runtime behaviors [9, 18]. The second challenge is data. Without a scalable evaluation method, there is no reliable way to filter high-quality projects from noisy open-source repositories, making large-scale dataset construction infeasible. Together, the absence of a deterministic evaluation framework makes it difficult to curate high-quality datasets or objectively assess model-generated games (Figure 1).

The Godot engine offers a unique opportunity to bridge this gap. Unlike engines that rely on binary formats and graphical interfaces [32], Godot adopts a fully text-based project format: its scene files (.tscn),

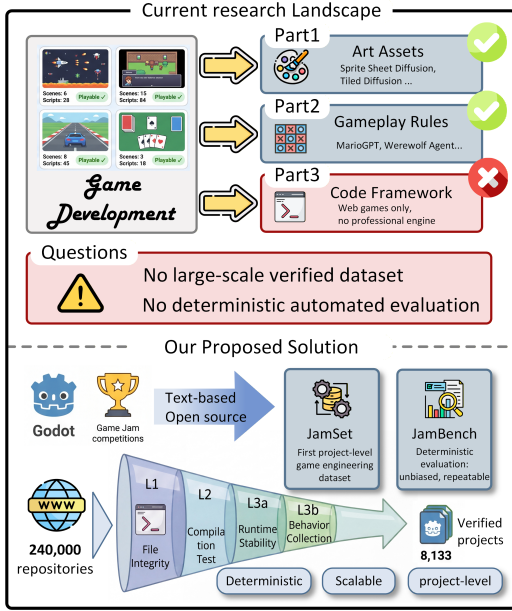


Figure 1 Research landscape and our proposed solution. Upper: three layers, with code framework largely unexplored. Lower: leveraging Godot, we build JamSet and JamBench with a deterministic verification pipeline.

script files (.gd), and project configuration (.godot) are all human-readable plain text [5], naturally compatible with LLM processing. Moreover, Godot is the fastest-growing open-source game engine, with steadily rising adoption in Game Jam competitions (Figure 3a). Game Jam competitions [31, 19, 21] provide an ideal data source: strict time constraints (48–72 hours) ensure entries are compact yet complete, and the open-source sharing tradition yields large-scale, freely accessible real-world data. Crucially, Godot’s headless mode allows games to run without a graphical interface, enabling fully automated execution and runtime behavior collection at scale.

Building on these properties, we construct JamSet and JamBench. We first design a deterministic verification pipeline on Godot’s headless mode, progressively checking file integrity, compilation correctness, runtime stability, and runtime behavior. This pipeline distills 8,133 verified projects from over 240,000 candidate repositories. Based on this, 300 are manually verified as the benchmark subset JamBench; the remaining 7,833 are processed into multi-turn training data as JamSet. JamBench covers theme-driven from-scratch generation (Task 1) and multi-granularity code completion (Task 2), evaluated through compilation pass rates, Structural Completeness Score (SCS) measuring static structural coverage, and Behavioral Alignment Score (BAS) measuring

runtime behavioral similarity to real projects.

We evaluate 9 frontier models and 5 Code Agent configurations on JamBench. Results reveal a capability cliff as project scale increases [18, 27, 46]: on Task 2, runtime pass rates drop from 80.4% on small projects to 5.7% on large ones (Task2a). **Code Agents substantially improve compilation pass rates yet yield no gains in runtime behavioral quality** [42, 2], indicating that the bottleneck lies in architectural design rather than syntactic correctness. Fine-tuning on JamSet validates the dataset’s effectiveness: the base model shows improved compilation rates and structural completeness, while also adopting human engineering practices such as input abstraction and global state management.

Our main contributions are: **(1)** The first project-level game code framework dataset and benchmark on a professional game engine. **(2)** A deterministic verification pipeline from file integrity to runtime behavior collection. **(3)** Systematic evaluation revealing a capability cliff at project scale and the limitation of Code Agents to syntactic repair over architectural design.

2 Related Works

Game Generation and Evaluation. AI-driven game creation has advanced rapidly. GameTileNet [4] and similar work focus on art asset generation, while MarioGPT [33] leverages LLMs for gameplay design. On the code framework front, V-GameGym [45] evaluates single-file game generation on Pygame, and OpenGame [17] builds an agentic framework for web games. On professional engines, GameDevBench [5] defines local modification tasks on Godot, AutoUE [43] and UnrealLLM [36] construct generation systems on Unreal Engine 5, and UniGen [39] and DreamGarden [10] explore multi-agent approaches on Unity and Unreal. These efforts have pushed game code generation toward professional engines, but existing work is either small in scale or relies on subjective evaluation (Table 1).

Game and Code Datasets. From a dataset perspective, existing game evaluation datasets are far smaller than what systematic research requires: GameDevBench [5] contains 132 tasks, OpenGame [17] 150 prompts, AutoUE [43] 20 tasks, and V-GameGym [45, 40] 2,219 single-file samples. In general code generation, evaluation has progressed from function-level (HumanEval [3, 22, 26], MBPP [1]) through class-level (ClassEval [9]) to repository-level (SWE-bench [18, 23, 24], DevEval [25], RepoBench [27],

Work	Engine	Task	Scale	Evaluation	Determ.	Scalable	Proj.-level	Dataset
V-GameGym	Pygame	Single-file	2,219	LLM judge	✗	✓	✗	✓
OpenGame	Phaser/JS	From-scratch	150	VLM judge	✗	✓	✗	✗
GameDevBench	Godot	Local edit	132	Test scripts	✓	✗	✗	✗
AutoUE	UE5	From-scratch	20	LLM judge	✗	✗	✓	✗
Ours	Godot	Both	8,133	Engine verify	✓	✓	✓	✓

Table 1 Comparison of related works. “Both” refers to from-scratch generation and multi-granularity completion.

CrossCodeEval [8, 37, 29], R2E-Eval [16], CodAgent-Bench [44]). However, none address the game domain, where unit tests cannot capture runtime interactive behaviors. Our pipeline filters 8,133 verified project-level game projects from 240K repositories.

Evaluation Methodology. Existing game evaluation methods fall into three categories. Hand-written test scripts (e.g., GameDevBench [5, 38, 47], ProxyWar [30]) [6] are precise and reproducible but expensive to scale. VLM/LLM scoring (e.g., OpenGame [17], V-GameGym [45], AutoUE [43]) [14] is scalable but subjective and hard to reproduce. Unit testing (e.g., SWE-bench [18, 35, 15], Big-CodeBench [46]) [25, 27, 12] is objective but unable to capture game-specific runtime behaviors. Our framework builds a deterministic pipeline on Godot’s headless engine from compilation to runtime behavior collection, without relying on any LLM judgment or manual annotation.

3 Dataset Construction

3.1 Game Jam Ecosystem and Data Sources

Game Jams are time-limited game development competitions in which participants build complete games from scratch within 48–72 hours around a given theme. These events are large in scale and entries are typically shared as open-source projects, forming a natural large-scale corpus of real-world game engineering artifacts. We collect data from three major platforms: Ludum Dare, itch.io, and Global Game Jam. After merging and deduplication, we obtain 37,588 repositories, from which 5,034 Godot engine projects are identified. Godot’s scene files (.tscn), script files (.gd), and project configuration (.godot) are all human-readable plain text with no binary serialization, making them naturally compatible with LLM text processing.

To establish filtering criteria for the dataset, we conduct a correlation analysis on 1,872 projects that have both Ludum Dare ratings and GitHub repositories. The Spearman correlation between game

code lines (excluding third-party plugins) and overall rating is $\rho = 0.445$ ($p < 0.001$), with a significant rating cliff below 1,200 lines. Based on this analysis, we set `game_lines` $\geq 1,200$ as the quality threshold and `addon_lines` $< 1,000$ as the plugin dependency limit. Data sources and correlation results are shown in Figure 3.

3.2 Data Collection and Filtering

The 5,034 Game Jam projects alone are insufficient for systematic research. We conduct large-scale searches across **GitHub** and additional jam platforms including **GMTK Game Jam**, **Godot Wild Jam**, **GitHub Game Off**, and **Brackeys Jam**, yielding approximately 240,000 candidate repositories. We apply the criteria from Section 3.1 as pre-filters (Godot 4.x, 2D, `game_lines` $\geq 1,200$, `addon_lines` $< 1,000$), retaining 37,638 projects. These are then passed through three levels of verification: L1 checks file integrity, L2 performs compilation verification, and L3a runs the game for 30 seconds to verify stability (technical details in Section 4.2). From over 240,000 candidates, 8,549 projects pass all three levels. Approximately 96% of candidates are filtered out due to missing files, compilation errors, version incompatibility, or runtime crashes, demonstrating the high noise level in open-source game repositories. Figure 2 illustrates the complete collection and filtering pipeline.

3.3 Data Annotation

We perform structured annotation on these projects to support evaluation and training.

manifest.json is extracted fully automatically through static analysis, containing each project’s script list, scene tree structure, input mappings, autoload configuration, and scene transition graph.

game_description.json is generated with LLM assistance, taking manifest information as input and producing gameplay descriptions and genre classifications. We collect 108 real Game Jam themes from real Game Jams platforms (Ludum Dare, Global Game Jam, GMTK), and use sentence-transformers

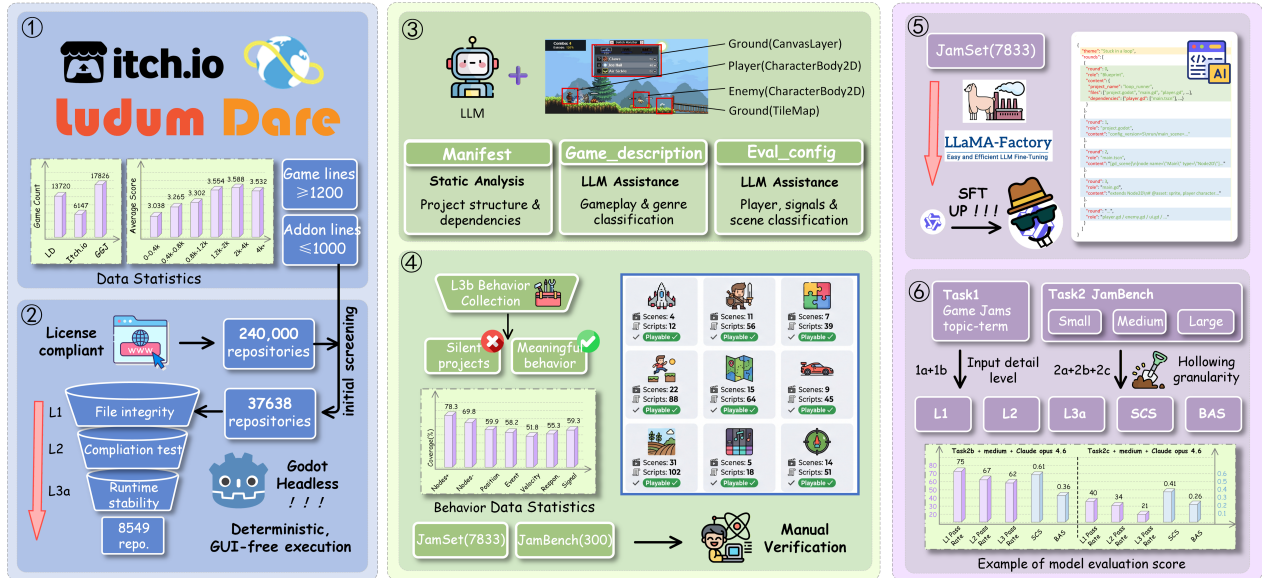


Figure 2 Overview of the GameJamBench pipeline. ① Data collection from Game Jam platforms. ② Multi-level filtering via Godot headless. ③ Structured annotation. ④ L3b behavior collection and dataset split. ⑤ Training data construction and SFT. ⑥ Evaluation tasks and metrics.

for embedding-based theme matching, retaining 89 themes.

eval_config.json is generated via LLM to identify each project’s player node, score and health tracking mechanisms, key signals, and the distinction between menu and gameplay scenes. This information drives precise input strategy generation during behavior collection (see Section 4.4).

asset_annotations.json uses a hybrid approach combining VLM and rule-based methods. We scan `res://` references and `ext_resource` declarations in `.gd`, `.tscn`, and `.tres` files. Image assets are annotated by VLM (with AtlasTexture regions cropped before annotation), while audio and font files are inferred from filenames and context. A total of 571,941 asset files are annotated. Note that all LLM-assisted annotations are used solely for data organization and training data construction; the verification pipeline involves no LLM judgment.

3.4 Dataset Statistics

After annotation, we run L3b behavior collection on all 8,549 projects (see Section 4.4), generating deterministic input strategies from **eval_config** and collecting runtime behavior data over 60 seconds of engine execution. Of these, 8,133 projects produce meaningful runtime behavior; 416 silent projects are excluded. These are divided into three tiers at 4K and 15K code lines: Small, Medium, and Large, span-

ning over 40 game genres, as shown in Figure 3c,d. We randomly sample S100/M100/L100 totaling 300 projects, each manually verified by playing for 3–5 minutes with a 100% pass rate, both to cross-validate the reliability of our automated pipeline and to ensure benchmark quality. The remaining 7,833 projects are reverse-engineered into multi-turn dialogue training data, with token counts ranging from 21K (Small) to 197K (Large). Detailed dataset statistics and training data construction are provided in Appendix C.

4 Benchmark Design

4.1 Task Definition

Task 1: Theme-driven generation. Given a Game Jam theme, the model must build a complete Godot project from scratch. Task 1a provides only a theme keyword (e.g., “Stuck in a loop”), while Task 1b additionally provides a gameplay description, isolating creative planning from engineering implementation. The model first outputs a project blueprint, then generates files in dependency order. The test set contains 50 real Game Jam themes, with experiments repeated three times.

Task 2: Multi-granularity code completion. Given a real game project with portions of code removed, the model must complete the missing parts. We define three granularity levels: 2a function-level (30–50% of function bodies cleared), 2b script-level (30–50% of

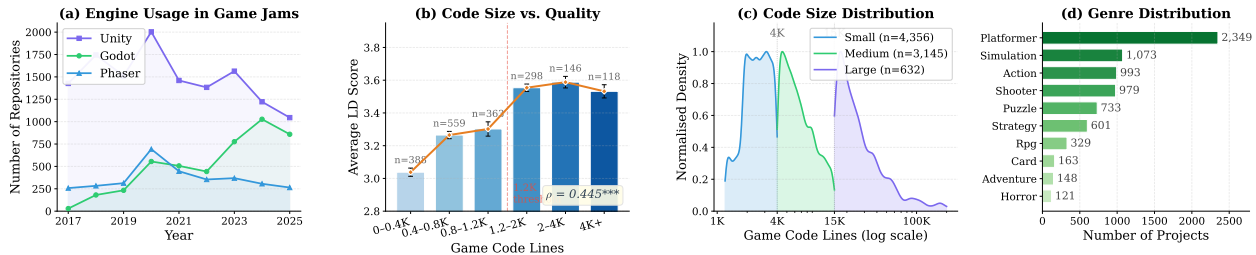


Figure 3 (a) Engine usage trends in Game Jam repositories (2017–2025); (b) Code size vs. Ludum Dare rating; a quality threshold at 1.2K lines is adopted based on Spearman $\rho=0.445$ ($p<0.001$). (c) Code size distribution across three tiers with boundary cutoffs at 4K and 15K lines. (d) Top 10 genre distribution across 8,133 projects.

.gd files removed), and 2c full-script-level (all .gd files removed). Scene files (.tscn) are fully preserved. The model receives the project blueprint and all retained file contents as context, and outputs completions sequentially. The test set consists of 300 benchmark projects, each with all three granularity levels.

4.2 Verification and Evaluation Pipeline

We build a four-level verification pipeline, fully deterministic and reproducible. During dataset construction, L1/L2/L3a are used for progressive filtering (Section 3.2), while L3b performs behavior collection and filters out silent projects (Section 3.4). During model evaluation, L1/L2/L3a serve as pass/fail tests, and L3b is used solely for data collection to compute BAS, not for filtering.

L1: File integrity. Verifies that `project.godot` exists and targets Godot 4.x, that the main scene is configured, and that all scripts and sub-scenes referenced in .tscn files are reachable. Projects with over 30% 3D content are excluded. References within the `addons` directory are handled separately to avoid false rejections.

L2: Compilation correctness. Runs Godot headless compilation to catch syntax errors, type errors, and missing resource references. Checks for essential code patterns: input handling and game loop are required, while collision detection, state management, and scene transition are recommended.

L3a: Runtime stability. Launches the game in headless mode for 30 seconds with no input injection, verifying no crash occurs. This level filters projects that crash on startup, time out during resource loading, or are incompatible with execution.

L3b: Runtime behavior collection. Relies on `eval_config` annotations to automatically generate a deterministic input strategy, then runs the game in the engine for 60 seconds while collecting multi-dimensional

behavior data. Technical details are provided in Section 4.4.

4.3 Evaluation Metrics

We evaluate game engineering along two dimensions: static structure and runtime behavior.

Structural Completeness Score (SCS). SCS measures the structural completeness of the generated code framework through static analysis across 7 dimensions: script count, scene count, input action count, function count, node count, non-empty function ratio, and signal usage. These dimensions capture the core structural aspects of Godot game projects. SCS is defined as:

$$SCS = \frac{1}{N} \sum_{i=1}^N \min\left(\frac{v_i}{v_i^{ref}}, 1.0\right) \quad (1)$$

where v_i is the generated value on dimension i , v_i^{ref} is the reference value, and $N = 7$. For Task 1, v_i^{ref} is the same-genre dataset mean; for Task 2, v_i^{ref} is the original project’s value. When $v_i = v_i^{ref} = 0$, the score is 1.0.

Behavioral Alignment Score (BAS). We select behavioral dimensions with non-zero coverage exceeding 50% across the dataset, yielding 7 numeric dimensions (nodes added, nodes removed, position changes, event count, velocity active frames, responsive actions) and 1 set dimension (signal trigger type overlap). Detailed coverage statistics by genre are provided in Appendix C.3. The per-dimension similarity is:

$$s_j^{num} = 1 - \frac{|b_j - b_j^{ref}|}{\max(b_j, b_j^{ref})}, \quad s_j^{sig} = \frac{|S_{gen} \cap S_{ref}|}{|S_{gen} \cup S_{ref}|} \quad (2)$$

When both $b_j = b_j^{ref} = 0$, $s_j^{num} = 1.0$; when both $S_{gen} = S_{ref} = \emptyset$, $s_j^{sig} = 1.0$. The overall BAS is:

$$BAS = \frac{1}{|D_{num}| + 1} \left(\sum_{j \in D_{num}} s_j^{num} + s_j^{sig} \right) \quad (3)$$

Cate.	Model	Task 1a (Theme Only)					Task 1b (Theme + Description)				
		L1	L2	L3a	SCS	BAS	L1	L2	L3a	SCS	BAS
LLM	Gemini 3.1 Pro	<u>100.0</u>	<u>85.3</u>	<u>78.7</u>	0.37	0.14	96.0	<u>62.7</u>	58.7	0.57	<u>0.31</u>
	Claude Opus 4.6	<u>100.0</u>	80.7	77.3	0.41	0.11	94.7	57.3	50.0	<u>0.66</u>	0.26
	GPT-5.4	94.7	77.3	77.3	<u>0.46</u>	<u>0.17</u>	94.7	66.0	<u>60.7</u>	0.63	<u>0.31</u>
	DeepSeek V4 Pro	97.3	82.7	72.7	0.28	0.09	93.3	57.3	52.7	0.44	0.17
	Kimi K2.5	92.0	79.3	72.0	0.39	0.16	92.0	60.7	51.3	0.46	0.21
	GLM-5	98.0	80.7	70.7	0.35	0.17	91.3	61.3	49.3	0.43	0.24
	Qwen3.5-397B	94.0	73.3	67.3	0.35	0.17	78.7	59.3	53.3	0.41	0.22
	Qwen3.5-27B	78.7	66.0	58.7	0.27	0.09	68.7	51.3	29.3	0.33	0.11
Qwen3.5-27B-SFT	88.7	70.7	62.0	0.34	0.21	86.0	61.3	34.7	0.41	0.23	
Agent	Claude Opus 4.6	<u>100.0</u>	<u>92.7</u>	<u>82.7</u>	<u>0.42</u>	0.13	<u>100.0</u>	82.7	<u>80.0</u>	<u>0.62</u>	<u>0.28</u>
	DeepSeek V4 Pro	<u>100.0</u>	93.3	84.0	0.34	0.12	<u>100.0</u>	80.0	76.7	0.49	0.14
	Kimi K2.5	<u>100.0</u>	90.0	81.3	0.31	<u>0.19</u>	<u>100.0</u>	78.0	77.3	0.37	0.17
	GLM-5	<u>100.0</u>	91.3	78.7	0.37	0.17	<u>100.0</u>	77.3	73.3	0.41	0.11
	Qwen3.5-397B	<u>100.0</u>	90.0	80.7	0.31	0.14	<u>100.0</u>	70.7	68.7	0.43	0.16

Table 2 Task 1 from-scratch generation results. L1/L2/L3a are pass rates (%). SCS and BAS are mean over 3 runs.

SCS Dimension	Ref.	BAS Dimension	Ref.
script_count	26.42	nodes_added	4720.31
scene_count	22.22	nodes_removed	4411.50
input_action	6.53	position_changes	131.18
function_count	128.62	event_count	33.38
node_count	201.55	velocity_active	102.17
non_empty_ratio	0.98	responsive_actions	4.73
signal_usage	58.02	signal_triggers	4.0

Table 3 Dataset reference values for Task 1 evaluation. SCS references are dataset means. BAS references are means from projects with rich runtime behavior. BAS dimensions are selected with >50% non-zero coverage.

For Task 2, references are the original project’s behavior data. For Task 1, references are dataset means (Table 3) with capped ratio $s_j = \min(b_j/b_j^{ref}, 1.0)$. Only non-zero dimensions contribute; if fewer than 2 are non-zero, one zero-valued dimension is included.

Model outputs are evaluated using five metrics: L1, L2, and L3a as pass/fail verification, and SCS and BAS as quantitative scores. The two scores are complementary: low SCS with passing compilation indicates minimal code, while high SCS with low BAS indicates structurally complete but behaviorally incorrect output.

4.4 L3b Behavior Collection

The core challenge is to perform meaningful interactions and collect behavior data for thousands of diverse games in a headless environment while ensuring full determinism. We address this with a three-layer architecture: (1) offline LLM preprocess-

ing generates `eval_config` from project structure, identifying player nodes, key signals, and scene classification; (2) rule-based input strategy generation produces deterministic action sequences from `eval_config` with no LLM calls; (3) runtime execution injects inputs and collects 7-dimension behavior data over 60 seconds, with menu navigation in the first 10 seconds and gameplay interaction in the remaining 50. Notably, the goal of behavior collection is not to play the game to completion, but to trigger as many active nodes as possible, producing rich behavioral signals for comparison between generated and reference projects. The entire pipeline guarantees determinism: the same `eval_config` always produces the same input strategy and runtime behavior. Technical details and the full algorithm are provided in Appendix D.

5 Experiments

5.1 Experimental Setup

We conduct two categories of evaluation experiments. The first is **Direct LLM**, where LLMs generate outputs directly. This category includes 9 models (Claude Opus 4.6, GPT-5.4, Gemini 3.1 Pro, Kimi K2.5, DeepSeek V4 Pro, Qwen3.5-397B, GLM-5, Qwen3.5-27B) and one fine-tuned model (Qwen3.5-27B-SFT), which is LoRA-fine-tuned on Qwen3.5-27B using Jam Set. Training hyperparameters are detailed in Appendix B. The second is **Code Agent**, using Claude Code as the agent framework to drive LLMs through iterative debugging. All models are evaluated on the full test sets of Task 1 and

Tier	Model	Level 2a (Function)					Level 2b (Script)					Level 2c (Full-script)				
		L1	L2	L3a	SCS	BAS	L1	L2	L3a	SCS	BAS	L1	L2	L3a	SCS	BAS
Small	Gemini 3.1 Pro	97	88	88	0.94	0.69	100	96	93	0.83	0.64	96	85	77	0.62	0.51
	Claude Opus 4.6	100	90	84	0.92	0.71	100	92	88	0.79	0.52	93	81	77	0.65	0.37
	GPT-5.4	96	84	81	0.97	0.64	100	91	88	0.85	0.46	90	78	75	0.70	0.35
	DeepSeek V4 Pro	92	85	81	0.93	0.57	99	93	93	0.73	0.51	92	80	72	0.58	0.42
	Kimi K2.5	93	82	80	0.93	0.60	100	95	90	0.76	0.59	90	79	74	0.57	0.44
	GLM-5	90	83	79	0.90	0.57	97	89	83	0.74	0.54	89	81	70	0.62	0.41
	Qwen3.5-397B	94	86	81	0.95	0.62	92	90	84	0.77	0.54	87	75	71	0.66	0.38
	Qwen3.5-27B	83	73	70	0.92	0.51	88	76	72	0.56	0.41	78	69	62	0.50	0.33
	Qwen3.5-27B-SFT	86	80	80	0.93	0.60	90	85	81	0.73	0.57	82	73	67	0.61	0.37
Medium	Gemini 3.1 Pro	61	50	32	0.80	0.46	72	59	49	0.54	0.35	43	32	19	0.37	0.23
	Claude Opus 4.6	56	53	44	0.82	0.51	75	67	62	0.61	0.36	40	34	23	0.41	0.26
	GPT-5.4	59	48	38	0.76	0.47	68	61	56	0.56	0.36	47	33	20	0.39	0.19
	DeepSeek V4 Pro	55	44	34	0.68	0.45	69	64	58	0.53	0.31	39	32	17	0.43	0.20
	Kimi K2.5	61	54	43	0.69	0.51	65	57	53	0.49	0.33	42	30	16	0.36	0.24
	GLM-5	47	42	37	0.66	0.47	67	60	45	0.47	0.27	40	26	23	0.32	0.19
	Qwen3.5-397B	56	48	41	0.67	0.50	65	53	53	0.51	0.29	37	31	19	0.29	0.25
	Qwen3.5-27B	32	29	17	0.52	0.46	48	46	23	0.28	0.19	25	13	6	0.15	0.08
	Qwen3.5-27B-SFT	43	31	26	0.61	0.45	52	44	38	0.40	0.26	33	20	14	0.22	0.17
Large	Gemini 3.1 Pro	32	14	7	0.53	0.50	39	18	13	0.43	0.30	9	4	1	0.29	0.01
	Claude Opus 4.6	28	17	12	0.50	0.52	41	22	16	0.45	0.31	15	9	2	0.23	0.06
	GPT-5.4	30	14	7	0.48	0.56	35	14	11	0.43	0.26	11	4	2	0.08	0.02
	DeepSeek V4 Pro	24	10	4	0.58	0.49	27	9	9	0.41	0.29	10	6	0	0.00	0.00
	Kimi K2.5	26	11	5	0.46	0.48	32	17	12	0.38	0.24	8	5	0	0.00	0.00
	GLM-5	23	13	9	0.44	0.44	29	22	14	0.39	0.26	8	4	1	0.21	0.04
	Qwen3.5-397B	19	11	5	0.56	0.48	25	17	13	0.40	0.18	6	3	0	0.00	0.00
	Qwen3.5-27B	3	0	0	0.00	0.00	11	4	2	0.27	0.04	0	0	0	0.00	0.00
	Qwen3.5-27B-SFT	12	4	2	0.45	0.51	19	13	8	0.33	0.11	3	0	0	0.00	0.00

Table 4 Task 2 Direct LLM results by Tier and granularity. L1/L2/L3a: pass rates (%). SCS/BAS: scores in [0, 1].

Task 2 (Section 4.1), reporting L1, L2, L3a, SCS, and BAS (Section 4.3). Token length statistics for training data and evaluation prompts are provided in Appendix C.

5.2 Direct LLM Results

Table 2 reports the performance of 9 LLMs on Task 1. Building games from scratch yields reasonable pass rates: L3a averages 70.7%. However, SCS and BAS scores are far lower, indicating that models tend to produce compilable but structurally minimal projects. When provided with gameplay descriptions (Task 1b), most models show improved SCS and BAS, but compilation pass rates decline, revealing a trade-off between design complexity and engineering capability. Qwen3.5-27B-SFT shows improvements over its base version in both compilation rate and SCS, validating the effectiveness of the training data.

Table 4 reports Task 2 results, broken down by Tiers and granularity (2a/2b/2c). Project scale is a bottleneck: all models exhibit a sharp performance cliff from Small to Large, consistent with findings in software engineering research. More detailed trend analysis is provided in Section 6.

5.3 Code Agent Results

Table 2 reports Code Agent performance on Task 1, and Table 6 reports results on Task 2. The Agent

mode improves pass rates across the board: on Task 1a, Claude+Agent raises L3a from 77.3% to 82.7%, and DeepSeek+Agent from 72.7% to 84.0%. However, SCS and BAS show minimal change: Claude+Agent achieves SCS of 0.42 (LLM: 0.41) and BAS of 0.13 (LLM: 0.11). More detailed analysis is provided in Section 6.

6 Analysis

6.1 Core Insights

Function-level completion demands more contextual precision. A counter-intuitive pattern emerges: despite removing more code, script-level completion (2b) achieves higher compilation pass rates than function-level (2a). Function-level completion requires precise restoration consistent with retained code in interfaces and logic, where a single incorrect function can break compilation. Script-level completion allows rewriting with internally consistent logic, free from existing code constraints. Interestingly, SCS and BAS show the opposite trend: 2a scores higher than 2b, as more original code is preserved. This suggests that completion granularity does not follow a simple difficulty progression, and function-level completion demands higher contextual precision. ClassEval [9] reports similar findings in a different domain.

Code Agent improvements are limited to compila-

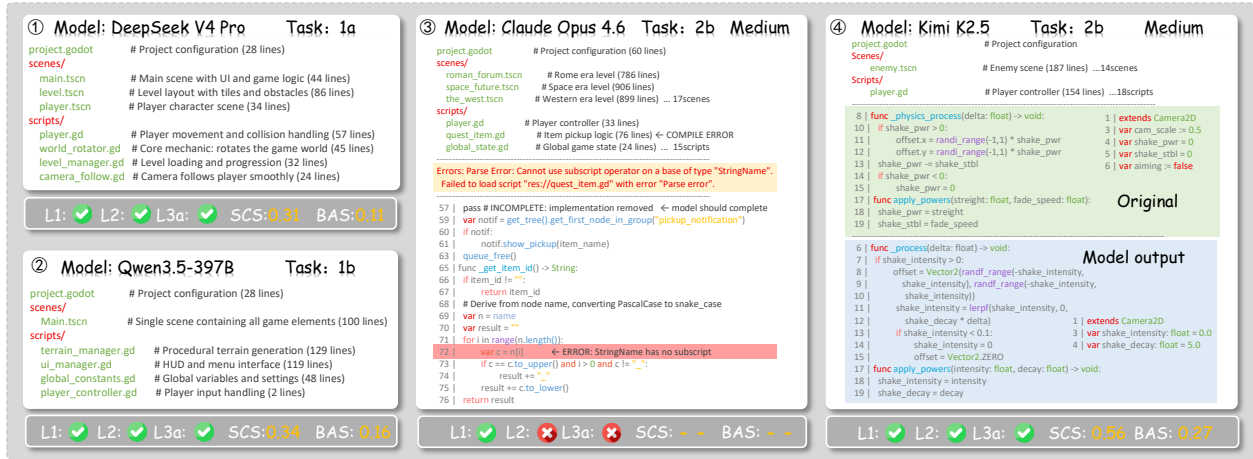


Figure 4 Representative case studies. Case 1 (12): Passing verification but with minimal functionality. Case 2 (3): Compilation failure. Case 3 (4): semantically drifted implementation breaking cross-file dependencies.

tion. Agents raise L3a pass rates by over 30 percentage points on average, yet SCS and BAS show no corresponding improvement. In Task 1, Agents improve compilation rates, but structural completeness and runtime behavior remain on par with direct LLM outputs, indicating that compilability alone is not a sufficient measure of game quality; Task 2 exhibits a consistent pattern (Appendix B). In contrast, the SCS gain from Task 1a to 1b shows that detailed gameplay planning can effectively guide the model to generate more complete game projects.

Engineering paradigm gap. Model-generated projects exhibit systematic differences from real projects. Most notably, real projects define an average of 6.9 input actions via abstract interfaces, while model-generated projects rely entirely on hard-coded key events (`input_action_count` near zero). Similarly, 76.3% of real projects use autoload scripts for global state management, while models concentrate logic in single scripts. These differences suggest that models learn to produce code that runs rather than proper game engineering practices. After fine-tuning on JamSet, Qwen3.5-27B-SFT shows significant improvement: `input_action_count` rises from 0.08 to 3.44, autoload usage from 55.1% to 77.1%, and scene transition usage from 18% to 49.4%, demonstrating that domain-specific training data guides models toward human-like engineering practices.

6.2 Case Study

We present three typical cases (Figure 4).

Case 1: Shortcut outputs. DeepSeek V4 Pro and Qwen3.5-397B both pass three verification levels on Task 1a, yet achieve low SCS scores. Notably, Qwen’s

project contains seemingly complex modules such as `terrain_manager` (129 lines) and `ui_manager` (119 lines), but the core `player_controller.gd` has only 2 lines of code, with the model failing to implement any actual game logic. This confirms that compilation pass rates alone cannot distinguish between a runnable shell and a real game, validating SCS and BAS as necessary complementary metrics.

Case 2: Lack of engine-specific knowledge. Claude Opus 4.6 demonstrates correct algorithm design in Task 2b, implementing a fully valid PascalCase-to-snake_case conversion, yet the code fails to compile due to a syntax issue with engine-specific types. The algorithm itself is correct in general-purpose programming, but GDScript imposes type constraints that differ from standard languages. Engine-specific Godot code constitutes a tiny fraction of existing training data, making it difficult for models to master these conventions. JamSet filters out substantial noise and organizes projects into a training-friendly structured format. Fine-tuning experiments show that training on JamSet notably improves model output quality, confirming the value of curated, structured domain data for bridging this knowledge gap.

Case 3: Cross-file semantic drift. Kimi K2.5 generates a functionally equivalent camera shake implementation in Task 2b that compiles and runs correctly. However, the rewritten code omits key state variables that other scripts rely on for camera zoom and aiming, causing silent behavioral degradation. This explains the gap between SCS (0.56) and BAS (0.27): structure appears complete but runtime interactions are incomplete, showing that models can replicate individual module intent but struggle to preserve cross-file implicit interfaces.

7 Conclusion

We present JamSet and JamBench, the first project-level game code framework dataset and benchmark on a professional game engine. Using a deterministic four-level verification pipeline on Godot’s headless mode, we distill 8,133 verified 2D game projects from over 240,000 repositories, and define from-scratch generation and multi-granularity completion tasks evaluated by SCS and BAS. Evaluation of frontier models reveals three findings: project scale is a hard capability bottleneck; Code Agent improvements are limited to compilation; and models exhibit systematic engineering paradigm gaps from human developers, which fine-tuning on JamSet effectively narrows.

Scope and Limitations

This work focuses on evaluating code frameworks rather than complete games with art and audio assets. Assessing such assets is inherently subjective and often dominates perceived game quality, undermining the reproducibility we require. Moreover, while asset generation and gameplay design have received considerable attention in prior work, code framework generation on professional engines remains unexplored. Since a well-structured code framework can support arbitrary asset integration and flexible design extensions, we argue that its verification is a meaningful and self-contained objective.

This work has the following limitations. First, the dataset focuses on the Godot game engine, which is growing rapidly as an open-source engine, but Unity, Unreal, and other engines still hold significant market share. The current dataset does not cover these engines, and we plan to continuously expand the application boundary in future work. Second, this work focuses on dataset construction and benchmark design. The exploration of model training serves only as preliminary validation of data effectiveness, without thorough ablation experiments. In the future, we plan to augment the dataset through project variants, repair projects that did not pass verification to expand the data scale, and train a domain-specific expert model for game engineering on this basis.

Ethics Statement

Data Sources and Licensing. All game projects used in this work are sourced from publicly available open-source repositories across platforms including Ludum Dare, itch.io, Global Game Jam, and GitHub. Only projects with open-source licenses are included. The

dataset contains no personal privacy information, and all project metadata have been stripped of developer identity markers.

Model Usage. Language models and vision-language models used during data annotation serve solely to generate structured annotations (gameplay descriptions, evaluation configurations, asset descriptions) and are not involved in any verification step. The four-level verification pipeline is fully automated and relies on no model judgment.

Open-Source Commitment. We will publicly release the complete dataset, evaluation framework, and tool code to facilitate research in game engineering automation.

References

- [1] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.
- [2] H. Che, X. He, Q. Liu, C. Jin, and H. Chen. Gamegen-x: Interactive open-world game video generation, 2024.
- [3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code, 2021.
- [4] Y.-C. Chen and A. Jhala. Gametilenet: A semantic dataset for low-resolution game art in procedural content generation, 2025.
- [5] W. Chi, Y. Fang, A. Yayavaram, S. Yayavaram, S. Karten, Q. A. Wei, R. Chen, A. Wang, V. Chen, A. Talwalkar, and C. Donahue. Gamedevbench: Evaluating agentic capabilities through game development, 2026.

- [6] R. Coppola, T. Fulcini, S. Manzi, and F. Strada. How to measure game testing: a survey of coverage metrics. In *Proceedings of the ACM/IEEE 8th International Workshop on Games and Software Engineering, GAS '24*, page 15–19, New York, NY, USA, 2024. Association for Computing Machinery.
- [7] F. Coutinho and L. Chaimowicz. On the challenges of generating pixel art character sprites using gans. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 18(1):87–94, Oct. 2022.
- [8] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In A. Oh, T. Nauemann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46701–46723. Curran Associates, Inc., 2023.
- [9] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.
- [10] S. Earle, S. Parajuli, and A. Banburski-Fahey. Dreamgarden: A designer assistant for growing games from a single prompt. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems, CHI '25*, New York, NY, USA, 2025. Association for Computing Machinery.
- [11] M. Farrokhi Maleki and R. Zhao. 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, Nov. 2024.
- [12] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, z. wang, S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 23247–23275, 2024.
- [13] C.-A. Hsieh, J. Zhang, and A. Yan. Sprite sheet diffusion: Generate game character for animation, 2025.
- [14] S. Hu, T. Huang, G. Liu, R. R. Kompella, F. Ilhan, S. F. Tekin, Y. Xu, Z. Yahn, and L. Liu. A survey on large language model-based game agents, 2025.
- [15] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *ArXiv*, abs/2312.13010, 2023.
- [16] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica. R2e: Turning any github repository into a programming agent environment. In *ICML*, 2024.
- [17] Y. Jiang, J. Hu, Q. Xiao, Y. Zheng, R. Ma, K. Feng, J. Han, T. Peng, K. Fan, M. Zhang, and X. Yue. Opengame: Open agentic coding for games, 2026.
- [18] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.
- [19] A. Kultima. Game jam natives?: The rise of the game jam era in game development cultures. In *Proceedings of the 6th International Conference on Game Jams, Hackathons and Game Creation Events, ICGJ 2021*, ACM International Conference Proceeding Series, pages 22–28, United States, Aug. 2021. ACM. Publisher Copyright: © 2021 ACM.; International Conference on Game Jams, Hackathons and Game Creation Events, ICGJ ; Conference date: 02-08-2021 Through 02-08-2021.
- [20] A. Kumarappan, P. A. Golnari, W. Wen, X. Liu, G. Ryan, Y. Sun, S. Fu, and E. Nallipogu. Devbench: A realistic, developer-informed benchmark for code generation models, 2026.
- [21] G. Lai, A. Kultima, F. Khosmood, J. Pirker, A. Fowler, I. Vecchi, W. Latham, and F. Fol Leymarie. Two decades of game jams. In *Proceedings of the 6th Annual International Conference on Game Jams, Hackathons, and Game Creation Events, ICGJ '21*, page 1–11, New York, NY, USA, 2021. Association for Computing Machinery.
- [22] J. Li, H. Deng, Y. Zhang, K. Zhang, T. Shao, T. Zhao, W. Wang, Z. Jin, G. Li, Y. Liu, Y. Fang, and Y. Dong. Realbench: A repo-level code generation benchmark aligned with real-world software development practices, 2026.

- [23] J. Li, G. Li, X. Zhang, Y. Dong, and Z. Jin. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories, 2024.
- [24] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, Y. Li, B. Gu, and M. Yang. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics.
- [25] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, Y. Li, B. Gu, and M. Yang. DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics.
- [26] K. Liu, Y. Pan, Y. Xiang, D. He, J. Li, Y. Du, and T. Gao. ProjectEval: A benchmark for programming agents automated evaluation on project-level code generation. In W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 20205–20221, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [27] T. Liu, C. Xu, and J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Learning Representations*, volume 2024, pages 47832–47850, 2024.
- [28] O. Madar and O. Fried. Tiled diffusion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2025.
- [29] S. Ouyang, D. HUANG, J. Guo, Z. Sun, Q. Zhu, and J. M. Zhang. Dscodibench: A realistic benchmark for data science code generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(38):32628–32636, Mar. 2026.
- [30] W. Peng, X. Wang, and Q. Wu. Proxywar: Dynamic assessment of llm code generation in game arenas, 2026.
- [31] M. J. Scott, G. Ghinea, and I. Hamilton. Promoting inclusive design practice at the global game jam: A pilot evaluation. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, page 1–4. IEEE, Oct. 2014.
- [32] L. Soni and A. Kaur. Merits and demerits of unreal and unity: A comprehensive comparison. In *2024 International Conference on Computational Intelligence for Green and Sustainable Technologies (ICCGST)*, pages 1–5, 2024.
- [33] S. Sudhakaran, M. González-Duque, M. Freiberger, C. Glanois, E. Najjarro, and S. Risi. Mariogpt: Open-ended text2level generation through large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 54213–54227. Curran Associates, Inc., 2023.
- [34] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (pcgml), 2018.
- [35] W. Tan, W. Zhang, X. Xu, H. Xia, Z. Ding, B. Li, B. Zhou, J. Yue, J. Jiang, Y. Li, R. An, M. Qin, C. Zong, L. Zheng, Y. Wu, X. Chai, Y. Bi, T. Xie, P. Gu, X. Li, C. Zhang, L. Tian, C. Wang, X. Wang, B. F. Karlsson, B. An, S. Yan, and Z. Lu. Cradle: Empowering foundation agents towards general computer control. *arXiv preprint arXiv:2403.03186*, 2024.
- [36] S. Tang, K. Zhao, L. Wang, Y. Li, X. Liu, J. Zou, Q. Wang, and X. Chu. UnrealLLM: Towards highly controllable and interactable 3D scene generation by LLM-powered procedural content generation. In W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, editors, *Findings of the Association for Computational Linguistics: ACL 2025*, pages 19417–19435, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [37] K. Vergopoulos, M. N. Müller, and M. Vechev. Automated benchmark generation for repository-level coding tasks, 2025.
- [38] G. Wang, Y. Xie, Y. Jiang, A. Mandlkar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

- [39] S. Wu, Y. Huang, C. Gao, D. Chen, Q. Zhang, Y. Wan, T. Zhou, X. Zhang, J. Gao, C. Xiao, et al. Unigen: A unified framework for textual dataset generation using large language models. *arXiv preprint arXiv:2406.18966*, 2024.
- [40] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang. Agentless: Demystifying llm-based software engineering agents, 2024.
- [41] Z. Xu, C. Yu, F. Fang, Y. Wang, and Y. Wu. Language agents with reinforcement learning for strategic play in the werewolf game. In *Proceedings of the 41st International Conference on Machine Learning, ICML'24*. JMLR.org, 2024.
- [42] J. Yang, C. E. Jimenez, A. Zhang, K. Lieret, J. Yang, X. Wu, O. Press, N. Muennighoff, G. Synnaeve, K. Narasimhan, D. Yang, S. Wang, and O. Press. Swe-bench multimodal: Do ai systems generalize to visual software domains? In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 2794–2829, 2025.
- [43] L. Yin, W. Cheng, Z. Qin, T. Huang, Y. Li, and G. Ding. Autoue: Automated generation of 3d games in unreal engine via multi-agent systems, 2026.
- [44] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics.
- [45] W. Zhang, J. Yang, R. Tao, L. Chai, S. Guo, J. Wu, X. Chen, G. Cui, N. Ding, X. Xu, H. Wei, and B. Zhou. V-gamegym: Visual game generation for code large language models, 2025.
- [46] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, B. Hui, N. Muennighoff, D. Lo, D. Fried, X. Du, H. de Vries, and L. V. Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025.
- [47] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, B. Hui, N. Muennighoff, D. Lo, D. Fried, X. Du, H. de Vries, and L. V. Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025.

Appendix

Appendix Overview

A Additional Related Work	13
B Additional Experiment Results	14
C Dataset and Training Details	14
C.1 Training Data Construction	14
C.2 Training Hyperparameters	14
C.3 BAS Dimension Coverage by Genre . .	15
C.4 Benchmark and Evaluation Statistics .	16
D L3b Technical Details	16
D.1 Three-Layer Architecture	16
D.2 Determinism Guarantees	17
D.3 Input Strategy Rule Mapping	17
D.4 Menu Navigation Challenges	17
D.5 Silent Project Analysis	18
E Training Data Example	21
F Evaluation Prompts	24
F.1 Task 1: From-Scratch Generation . . .	24
F.2 Task 2: Code Completion	25
F.3 Eval Config Generation	26

A Additional Related Work

In the main text we primarily compare with game-domain benchmarks (Table 1). To further clarify our positioning in the broader code engineering evaluation landscape, we provide a detailed comparison with general-purpose code benchmarks here.

SWE-bench [18] collects 2,294 test tasks from 12 Python repositories (with a training set of 19,000 tasks across 37 repositories), requiring models to generate patches that fix real GitHub issues, verified through existing test suites. In terms of dataset construction, SWE-bench relies on GitHub issues and corresponding pull requests, with each task requiring developer-written test cases and manual environment configuration; our data is automatically filtered from 240,000 repositories without per-project

manual setup. In terms of evaluation, SWE-bench executes unit tests and reports pass/fail rates; we collect runtime behavior through the game engine without relying on pre-written test cases. In terms of task type, SWE-bench focuses on local repair; we extend beyond local repair to include from-scratch generation and multi-granularity completion.

DevEval [25] is manually annotated by 13 developers, containing 1,874 code generation samples from 117 Python repositories that simulate developers writing code in real repositories, verified through unit tests. In terms of construction, DevEval relies entirely on manual annotation, which is costly and difficult to scale; our pipeline is fully automated, producing a dataset over 4× larger. In terms of evaluation dimensions, DevEval reports Pass@1 and other pass/fail metrics; we additionally provide SCS and BAS as continuous indicators that capture cases where code compiles but lacks functionality. Furthermore, DevEval focuses on function-level and file-level generation, while we evaluate complete project generation and completion.

BigCodeBench [46] contains 1,140 function-level code generation tasks emphasizing diverse third-party library calls and complex instructions, verified through execution of test cases. In terms of construction, BigCodeBench tasks are manually designed with hand-written test cases; our test data comes from real game projects without requiring hand-written tests. In terms of granularity, BigCodeBench operates at the function level (single function bodies); we operate at the project level. BigCodeBench similarly reports pass/fail for function correctness.

ProjectEval [26] contains 20 tasks with 284 test cases, constructed by LLM generation with human review. Evaluation verifies functional correctness by simulating user interactions through test cases. In terms of construction, ProjectEval tasks are LLM-generated then human-reviewed, with scale limited by review cost (20 tasks); our data comes from real Game Jam projects, automatically filtered to 8,133 projects through a deterministic pipeline without per-project manual review. In terms of evaluation, ProjectEval relies on pre-written test scripts that simulate user interaction, requiring manual test design for each task; our evaluation requires no per-project test cases. Table 5 provides a detailed comparison.

DevBench [20] is a code completion benchmark containing 1,800 instances across 6 programming languages and 6 task categories. Its evaluation combines three methods: functional correctness checks, similarity metrics, and LLM-judge scoring. In terms of construction, DevBench extracts task types from

real developer telemetry, then generates instances synthetically with human review, making the process semi-automated; our data comes directly from real game projects. In terms of evaluation, DevBench incorporates LLM-judge as part of its assessment, providing richer quality dimensions but at the cost of reproducibility, as LLM scoring may vary across models or over time. In terms of granularity, DevBench focuses on code completion level (line/function); we evaluate complete project generation and completion.

Among mainstream benchmarks in both game and general code domains, our work offers differentiated advantages in scale, automation, and evaluation dimensions.

B Additional Experiment Results

Table 6 presents the complete Task 2 results for both Direct LLM and Code Agent experiments. Code Agent experiments use Claude Code as the agent framework, with ccswitch replacing the underlying model to drive Claude, DeepSeek, and Qwen3.5-397B through iterative debugging.

From the perspective of completion granularity, Task 2 exhibits the same counter-intuitive pattern: script-level completion (2b) achieves higher compilation pass rates than function-level (2a), regardless of whether LLM or Code Agent is used. In the Small tier, Claude LLM achieves 88% L3a on 2b versus 84% on 2a; Claude+Agent achieves 94% on 2b versus 90% on 2a. This trend is more pronounced in the Medium and Large tiers, further validating the finding in Section 6: function-level completion demands higher contextual precision than script-level completion, consistent with the granularity effect reported by ClassEval [9] in class-level code generation.

From the perspective of Agent improvement, Agents significantly boost compilation pass rates across all tiers and granularities. In the Medium tier 2b, Claude’s L3a rises from 62% to 76%, DeepSeek from 58% to 73%, and Qwen from 53% to 74%. However, SCS and BAS remain largely unchanged: Claude+Agent achieves 0.80 SCS on Medium 2a (LLM: 0.82) and 0.52 BAS (LLM: 0.51). This is fully consistent with Task 1 findings: Agent debugging focuses on fixing compilation errors and file references at the syntactic level, with no meaningful improvement in structural completeness or runtime behavioral quality. AgentCoder [15] reports similar findings in general code generation: multi-agent iterative testing and optimization effectively improve test pass rates, but improvements are concentrated at the surface correctness level in scenarios requiring

deep semantic understanding. Our results extend this observation to the game engineering domain: compilability is merely the minimum quality threshold, while scene structure design, cross-file logic coordination, and runtime interactive behavior are the true quality indicators. Current Agent repair mechanisms focus on eliminating compilation errors without reaching these higher-level engineering quality dimensions.

We therefore suggest that future Code Agent development should incorporate domain-specific knowledge into the iterative workflow, such as game engine engineering conventions, scene organization paradigms, and cross-file interface contracts, rather than checking only compilability at each iteration. When Agent feedback signals expand from “does it compile” to “does it follow domain engineering practices,” improvements in project quality can move beyond syntactic repair.

C Dataset and Training Details

C.1 Training Data Construction

Each of the 7,833 training projects is reverse-engineered into a multi-turn dialogue sample that simulates building a game from scratch: the first turn generates a project blueprint from a theme, and subsequent turns generate files in dependency-topological order. The following processing rules are applied:

GDScript files (.gd): Fully preserved with injected @asset semantic descriptions for referenced art and audio assets.

Scene files (.tscn): Complete node trees and properties retained; engine-generated uids and non-semantic binary data removed.

Resource files (.tres): Retained if under 10K tokens.

Excluded: .import and .cfg files. The resulting training data ranges from a median of 21K tokens for Small projects to 197K for Large. A complete example is provided in Appendix E. See Figure 5e for the token length statistics of the training data.

C.2 Training Hyperparameters

Table 8 lists the hyperparameters used for fine-tuning Qwen3.5-27B-SFT. We use LoRA with rank 8 on all linear layers, trained for 3 epochs on JamSet-Small and JamSet-Medium (7,300 samples). Larger-tier projects exceed the 32K token cutoff and are excluded from training.

Benchmark	Domain	Scale	Granularity	Construction	Verification	Det.	Scal.	Runtime
SWE-bench	Python	2,294	Repo repair	Issue+PR	Unit test	✓	✗	✗
DevEval	Python	1,874	Func/File	Manual	Unit test	✓	✗	✗
DevBench	Multi-lang	1,800	Line/Func	Telemetry+Synth	LLM judge	✗	✓	✗
BigCodeBench	Python	1,140	Function	Manual	Execution	✓	✗	✗
ProjectEval	Python	20	Project	LLM+Human	Interaction sim.	✓	✗	✓
Ours	Godot	8,133	Project	Auto pipeline	Engine headless	✓	✓	✓

Table 5 Comparison with code engineering benchmarks. Det. = Deterministic. Scal. = Scalable (no per-task manual annotation or test writing). Runtime = Evaluates runtime behavior.

Tier	Type	Model	Level 2a (Function)					Level 2b (Script)					Level 2c (Full-script)					
			L1	L2	L3a	SCS	BAS	L1	L2	L3a	SCS	BAS	L1	L2	L3a	SCS	BAS	
Small	LLM	Gemini 3.1 Pro	97	88	88	0.94	0.69	100	96	93	0.83	0.64	96	85	77	0.62	0.51	
		Claude Opus 4.6	100	90	84	0.92	0.71	100	92	88	0.79	0.52	93	81	77	0.65	0.37	
		GPT-5.4	96	84	81	0.97	0.64	100	91	88	0.85	0.46	90	78	75	0.70	0.35	
		DeepSeek V4 Pro	92	85	81	0.93	0.57	99	93	93	0.73	0.51	92	80	72	0.58	0.42	
		Kimi K2.5	93	82	80	0.93	0.60	100	95	90	0.76	0.59	90	79	74	0.57	0.44	
		GLM-5	90	83	79	0.90	0.57	97	89	83	0.74	0.54	89	81	70	0.62	0.41	
		Qwen3.5-397B	94	86	81	0.95	0.62	92	90	84	0.77	0.54	87	75	71	0.66	0.38	
		Qwen3.5-27B	83	73	70	0.92	0.51	88	76	72	0.56	0.41	78	69	62	0.50	0.33	
	Qwen3.5-27B-SFT	86	80	80	0.93	0.60	90	85	81	0.73	0.57	82	73	67	0.61	0.37		
	Agent	Claude Opus 4.6	100	96	90	0.93	0.72	100	98	94	0.80	0.53	100	92	85	0.66	0.38	
		DeepSeek V4 Pro	100	94	88	0.94	0.58	100	97	95	0.74	0.52	100	90	80	0.59	0.43	
		Qwen3.5-397B	96	91	85	0.93	0.66	100	96	92	0.78	0.56	99	87	77	0.52	0.34	
	Medium	LLM	Gemini 3.1 Pro	61	50	32	0.80	0.46	72	59	49	0.54	0.35	43	32	19	0.37	0.23
			Claude Opus 4.6	56	53	44	0.82	0.51	75	67	62	0.61	0.36	40	34	23	0.41	0.26
GPT-5.4			59	48	38	0.76	0.47	68	61	56	0.56	0.36	47	33	20	0.39	0.19	
DeepSeek V4 Pro			55	44	34	0.68	0.45	69	64	58	0.53	0.31	39	32	17	0.43	0.20	
Kimi K2.5			61	54	43	0.69	0.51	65	57	53	0.49	0.33	42	30	16	0.36	0.24	
GLM-5			47	42	37	0.66	0.47	67	60	45	0.47	0.27	40	26	23	0.32	0.19	
Qwen3.5-397B			56	48	41	0.67	0.50	65	53	53	0.51	0.29	37	31	19	0.29	0.25	
Qwen3.5-27B			32	29	17	0.52	0.46	48	46	23	0.28	0.19	25	13	6	0.15	0.08	
Qwen3.5-27B-SFT		43	31	26	0.61	0.45	52	44	38	0.40	0.26	33	20	14	0.22	0.17		
Agent		Claude Opus 4.6	88	76	70	0.80	0.52	88	80	76	0.62	0.37	62	57	52	0.42	0.27	
		DeepSeek V4 Pro	85	77	68	0.69	0.46	89	78	73	0.54	0.32	58	58	56	0.44	0.21	
		Qwen3.5-397B	78	74	70	0.68	0.47	84	78	74	0.50	0.27	62	58	54	0.37	0.29	
Large		LLM	Gemini 3.1 Pro	32	14	7	0.53	0.50	39	18	13	0.43	0.30	9	4	1	0.29	0.01
			Claude Opus 4.6	28	17	12	0.50	0.52	41	22	16	0.45	0.31	15	9	2	0.23	0.06
	GPT-5.4		30	14	7	0.48	0.56	35	14	11	0.43	0.26	11	4	2	0.08	0.02	
	DeepSeek V4 Pro		24	10	4	0.58	0.49	27	9	9	0.41	0.29	10	6	0	0.00	0.00	
	Kimi K2.5		26	11	5	0.46	0.48	32	17	12	0.38	0.24	8	5	0	0.00	0.00	
	GLM-5		23	13	9	0.44	0.44	29	22	14	0.39	0.26	8	4	1	0.21	0.04	
	Qwen3.5-397B		19	11	5	0.56	0.48	25	17	13	0.40	0.18	6	3	0	0.00	0.00	
	Qwen3.5-27B		3	0	0	0.00	0.00	11	4	2	0.27	0.04	0	0	0	0.00	0.00	
	Qwen3.5-27B-SFT	12	4	2	0.45	0.51	19	13	8	0.33	0.11	3	0	0	0.00	0.00		
	Agent	Claude Opus 4.6	46	37	28	0.51	0.53	50	40	35	0.46	0.32	33	23	21	0.14	0.13	
		DeepSeek V4 Pro	42	37	31	0.59	0.44	50	38	33	0.41	0.30	27	19	13	0.11	0.09	
		Qwen3.5-397B	40	31	26	0.48	0.51	46	37	30	0.44	0.25	28	20	17	0.13	0.07	

Table 6 Task 2 results for Direct LLM and Code Agent. L1/L2/L3a: pass rates (%). SCS/BAS: scores in [0, 1].

C.3 BAS Dimension Coverage by Genre

Table 9 reports the non-zero coverage rate of each BAS dimension across game genres. Platformer

games exhibit the highest average non-zero dimensions (4.8), driven by rich physics interactions, while card games show the lowest (2.5) due to minimal

Base Class	Count	Typical Role
Node2D	22,777	2D game objects
Node	21,786	General logic / managers
Control	14,536	UI elements
Area2D	10,660	Collision / trigger zones
CharacterBody2D	8,155	Player / NPC characters
Resource	4,647	Data resources
CanvasLayer	4,622	UI layers / HUD
Node3D	4,091	3D objects (mixed projects)
RefCounted	2,861	Pure logic classes
Button	2,489	Button widgets

Table 7 Top 10 script base classes (**extends**) across all projects, reflecting the structural composition of 2D game development in Godot.

Hyperparameter	Value
Base model	Qwen3.5-27B
Fine-tuning method	LoRA
LoRA rank	8
LoRA target	All linear layers
DeepSpeed stage	ZeRO-3
Attention	FlashAttention-2
Hardware	8 × NVIDIA H200 (141GB)
Precision	FP8 + BF16
Learning rate	1e-4
LR scheduler	Cosine
Warmup ratio	0.1
Epochs	3
Cutoff length	32,768
Per-device batch size	1
Gradient accumulation	2
Effective batch size	16
Gradient checkpointing	Enabled
Training data	JamSet-Small + JamSet-Medium (4,258 + 3,042 = 7,300 samples)
Training data format	ShareGPT multi-turn dialogue

Table 8 Fine-tuning hyperparameters for Qwen3.5-27B-SFT.

spatial movement. The overall average of 3.9 non-zero dimensions per project confirms that every game exhibits meaningful behavior across multiple dimensions.

C.4 Benchmark and Evaluation Statistics

Figure 5 summarizes the benchmark test set. (a) Word cloud of 50 Game Jam themes used in Task 1, sourced from 5 platforms with 89 themes matched to dataset projects. (b) First-round prompt token counts: Task 1a averages 352 tokens (theme only), while Task 1b averages 741 tokens with the additional gameplay description contributing 364 tokens. (c) Benchmark project scale distribution across three tiers, with median code lines of 2,468 (Small), 5,782 (Medium), and 23,180 (Large). (d) Code removal

ratios for Task 2 completion granularities: function-level (2a) removes 30–50% of functions, script-level (2b) removes 30–50% of scripts, and full-script (2c) removes 100% of script content.

D L3b Technical Details

D.1 Three-Layer Architecture

Layer 1: Offline preprocessing. Structural information is extracted from `manifest.json` (script extends and function signatures, scene node trees, input mappings) and processed by an LLM to generate `eval_config.json`. This identifies player node type and path, score and health tracking mechanisms, key game signals (e.g., `enemy_died`, `coin_collected`),

Genre	N	nodes+	nodes-	pos	event	vel	resp	signal	avg
platformer	2,349	74	70	85	62	92	79	60	4.8
simulation	1,073	80	63	37	49	35	33	72	3.0
action	993	75	69	70	61	94	63	76	4.3
shooter	979	79	72	62	60	88	67	83	4.2
puzzle	733	83	76	44	60	14	32	52	3.3
strategy	601	84	69	31	52	7	30	24	3.0
rpg	329	80	65	58	55	34	51	72	3.8
card	163	83	67	23	50	11	11	62	2.5
adventure	148	78	76	39	57	82	34	65	3.2
horror	121	83	75	44	61	61	45	27	3.7
...									
All	8,133	78	70	60	58	51	55	59	3.9

Table 9 Non-zero coverage rate (%) of each BAS dimension across game genres. “avg” is the average number of non-zero dimensions per project. All numeric dimensions exceed 50% overall coverage except velocity (39%) and signal triggers (34%); the latter is included as a set-based dimension capturing Godot’s signal-driven architecture.

menu versus gameplay scene classification, and expected behavior patterns (e.g., jumping and movement for platformers, shooting for shooters). This is a one-time offline annotation with fixed results. The LLM prompt template is provided in Appendix F.

Layer 2: Input strategy generation. From `eval_config`’s expected behaviors, input actions, and game type, `input_strategy.json` is generated through pure rule-based transformation with no LLM calls. It contains three components: input mode (keyboard/mouse/mixed), a deterministic action sequence (mapping expected behaviors to concrete actions), and a menu navigation strategy (targeting gameplay scenes identified in `eval_config`).

Layer 3: Runtime deterministic execution. As shown in Algorithm 1, execution proceeds in two phases. The menu navigation phase (first 10 seconds) scans `Button` nodes, reads their positions, and attempts to bypass menus through a three-level fallback: emit `pressed` signal, mouse click at `Button.global_position`, keyboard Enter/Space. Upon detecting a scene change to the target gameplay scene, execution advances to the next phase. The gameplay input phase (remaining 50 seconds) executes the deterministic action sequence according to the input mode, collecting all seven BAS dimensions throughout.

D.2 Determinism Guarantees

The same `eval_config` always produces the same `input_strategy`, and runtime execution follows a fixed order. Three safeguards prevent pause-related disruptions: the collection script’s `process_mode` is set to `ALWAYS` (unaffected by game pausing),

`SceneTree.paused` is checked and cleared every frame, and dangerous actions (pause, escape, quit) are excluded from input sequences.

D.3 Input Strategy Rule Mapping

The input strategy generator maps game types identified in `eval_config` to concrete action sequences through a deterministic rule table. Table 10 shows the mapping for common game types. For each game type, the generator selects an input mode and constructs a cyclic action sequence that covers the game’s expected interactions. The action sequence length is fixed at 600 frames (10 seconds at 60 FPS), repeating cyclically for the full 50-second gameplay phase. When `eval_config` specifies custom input actions (e.g., `move_left`, `jump`), these override the default mappings.

D.4 Menu Navigation Challenges

Menu bypass is a significant technical challenge because Game Jam games have no standardized menu structure. Some games start directly in gameplay, others have a single “Play” button, and many feature multi-layered menus with settings, credits, and level selection screens. We identified three categories of menu structures in the dataset:

No menu (31.2%): The main scene is the gameplay scene itself. L3b detects this when `eval_config.menu_scenes` is empty or the main scene appears in `gameplay_scenes`, and skips directly to the input phase.

Simple menu (52.4%): A single screen with `Button` nodes labeled “Play”, “Start”, or similar. The three-level fallback handles these reliably.

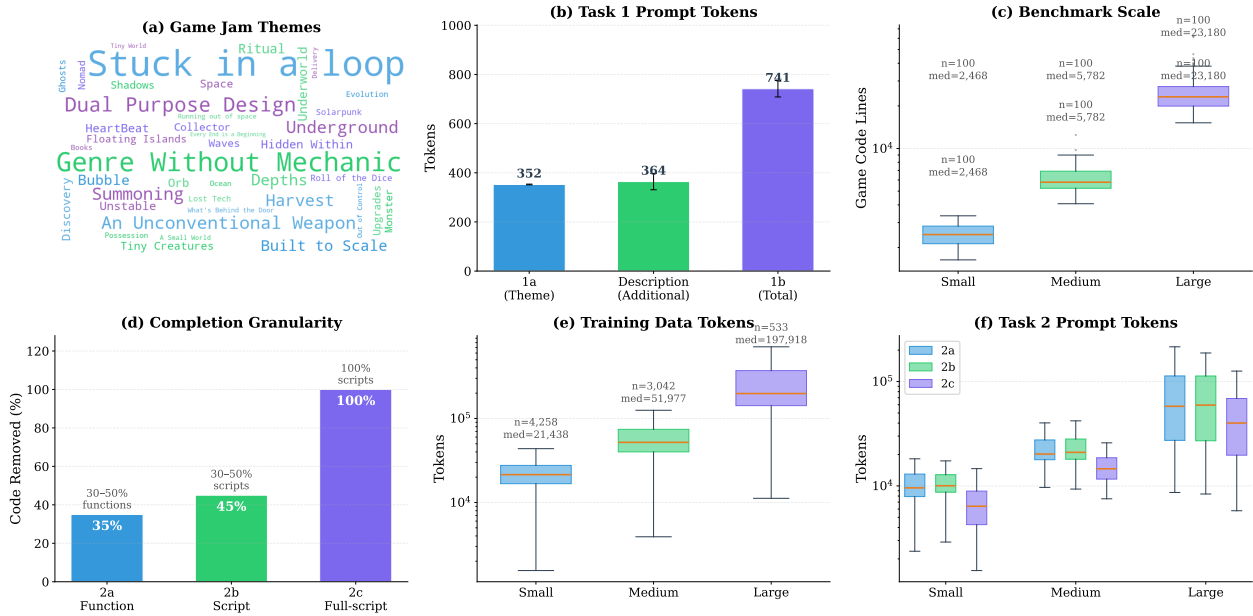


Figure 5 Benchmark and evaluation statistics. (a) Game Jam theme word cloud for Task 1. (b) First-round prompt token distribution. (c) Benchmark project scale by tier. (d) Code removal ratio by completion granularity.

Complex menu (16.4%): Multi-screen menus, animated transitions, or custom UI elements (e.g., clickable sprites instead of Button nodes). These require all three fallback levels and sometimes multiple attempts.

The three-level fallback is designed to handle this diversity:

Level 1: Signal emission. Directly emit the `pressed` signal on detected Button nodes. This is the most reliable method as it bypasses visual layout issues, but fails when the game uses custom click handlers instead of Godot’s built-in Button signals.

Level 2: Mouse click. Read the Button’s `global_position` and simulate a mouse click at that coordinate. This handles custom click handlers but can fail when buttons are obscured by overlapping UI elements or when the button position is not yet computed (e.g., during animated transitions).

Level 3: Keyboard fallback. Inject Enter and Space key events, which many games bind to menu confirmation. This serves as a last resort when no clickable buttons are found or when the game uses keyboard-only navigation.

After each attempt, L3b checks whether the current scene has changed to one listed in `eval_config.gameplay_scenes`. If no transition occurs after exhausting all buttons and fallback levels, the system proceeds with input injection in the current

scene, as some games may use in-scene transitions rather than scene changes.

D.5 Silent Project Analysis

Of the 8,549 projects that pass L3a (runtime stability), 416 (4.9%) produce no meaningful behavioral change during the 60-second L3b collection window. We manually inspected a random sample of 50 silent projects and identified three primary causes:

Visual-feedback-only games (44%): Games where all interaction feedback is purely visual (animations, particle effects, shader changes) with no state changes detectable through node properties. Examples include rhythm games where notes scroll visually but score is tracked only in shaders, and drawing games where player input creates visual output without modifying node tree structure.

Custom input systems (32%): Games that implement their own input handling bypassing Godot’s `InputAction` system. These games listen for raw `InputEventKey` or `InputEventMouse` events with specific scan codes that our input strategy does not cover, or use custom gesture recognition systems.

Passive or cutscene-style games (24%): Games with minimal player agency, such as interactive fiction where progression requires specific text input, auto-playing narrative sequences, or procedural art generators that run autonomously.

Algorithm 1: L3b Deterministic Behavior Collection

```
Input : eval_config  $E$ , input_strategy  
         $S = \{\text{mode, actions, menu\_targets}\}$   
Output: event_stream  
         $B = \{\text{pos, vel, sig, score, health, node, scene}\}$   
// Phase 1: Menu Navigation ( $t \in [0, T_{\text{menu}}]$ )  
1 buttons  $\leftarrow \{x \in \text{SceneTree} \mid x.\text{type} = \text{Button}\}$ ;  
2 Sort buttons by keyword priority (play > start > begin >  
   ...);  
3 for each button  $x$  in buttons do  
4   for  $f$  in  $\{\text{emit\_pressed, mouse\_click}(x.\text{pos}),$   
    $\text{key\_inject}(\text{Enter})\}$  do  
5      $f(x)$ ;  
6     if  $\text{current\_scene} \in E.\text{gameplay\_scenes}$  then  
7       goto Phase 2;  
  
// Phase 2: Gameplay Interaction  
   ( $t \in [T_{\text{menu}}, T_{\text{max}}]$ )  
8 for  $t = 0, 1, \dots, T_{\text{max}}$  do  
9    $a \leftarrow S.\text{actions}[t \bmod |S.\text{actions}|]$ ;  
10  if  $S.\text{mode} = \text{keyboard}$  then  
11     $\text{inject}(\text{ActionEvent}(a), \text{KeyEvent}(\text{keymap}(a)))$ ;  
12  else if  $S.\text{mode} = \text{mouse}$  then  
13     $\text{targets} \leftarrow \{x \in \text{SceneTree} \mid \text{is\_interactive}(x)\}$ ;  
14     $\text{inject}(\text{MouseClicked}(\text{targets}[t \bmod |\text{targets}|].\text{pos}))$ ;  
15  else  
16     $\text{inject}(\text{ActionEvent}(a))$ ;  
17    if  $t \bmod N = 0$  then  
18       $\text{inject}(\text{MouseClicked}(\text{next\_target}.\text{pos}))$ ;  
19   $B[t] \leftarrow$   
    $\text{snapshot}(\text{pos, vel, sig, score, health, nodes, scenes})$ ;  
20  if  $\text{SceneTree}.\text{paused}$  then  
21     $\text{SceneTree}.\text{paused} \leftarrow \text{false}$ ; // Unpause  
    immediately  
22 return  $B$ 
```

These 416 projects are excluded from the final dataset, as the absence of detectable behavioral signals makes BAS computation undefined. This exclusion does not bias the dataset toward simpler games: silent projects span all three tiers (Small: 5.1%, Medium: 4.6%, Large: 4.7%) and all major genres, indicating that silence is a property of the game’s interaction paradigm rather than its complexity.

Game Type	Mode	Action Sequence
platformer	keyboard	left/right movement (60%), jump (25%), attack (15%), direction alternation period: 2–3s
shooter	mixed	WASD movement (50%), mouse clicks at interactive targets (30%), attack action (20%)
puzzle	mouse	click interactive nodes in scene tree order, drag gestures between clickable targets
strategy	mouse	click UI elements and interactive nodes, periodic keyboard confirm (Enter/Space)
rpg	keyboard	directional movement (40%), interact (30%), attack (20%), menu navigation (10%)
card	mouse	click card-like nodes, drag between positions, periodic confirm clicks
other	mixed	WASD movement + periodic mouse clicks at scene center and interactive nodes

Table 10 Default input strategy mapping by game type. Custom input actions from `eval_config` override these defaults when available.

E Training Data Example

The following shows a representative training sample from JAMSET. Each sample is a multi-turn dialogue that simulates the complete process of building a game project from a Game Jam theme: `system prompt` defines technical requirements, `user instructions` request file generation in dependency order, and `model outputs` produce complete file contents (~4,200 lines).

Turn 0 [system]

```
You are an expert indie game developer competing in a Game Jam.
Technical requirements (Godot 4.x):
- Valid project.godot with config_version=5
- Scene files .tscn, scripts .gd; Main scene under run/main_scene
- No external plugins; Asset comments: # @asset: type, desc, size, style
```

Turn 1 [user]

```
The Game Jam theme is: "Stuck in a loop". Design a complete 2D game
and output a blueprint.
```

Turn 2 [assistant] — Blueprint (78 lines)

```
{ "game_name": "loop-runner", "genre": "action-platformer",
  "game_description": "A platformer where the level loops infinitely.
  The player must break the cycle by finding hidden triggers.",
  "core_mechanics": ["jump", "dash", "loop counter",
    "world mutation", "collectibles"],
  "files": ["project.godot", "scenes/main_menu.tscn",
    "scenes/game_world.tscn", "scenes/player.tscn",
    "scenes/enemy.tscn", "scenes/collectible.tscn",
    "scenes/hud.tscn", "scripts/player.gd",
    "scripts/enemy.gd", ...],
  "dependencies": {
    "game_world.tscn": ["player.tscn", "enemy.tscn",
      "hud.tscn"], ...} }
```

Turn 3 [user]

```
Now generate the file: project.godot
```

Turn 4 [assistant] — project.godot (68 lines)

```
config_version=5
[application]
config/name="loop-runner"
run/main_scene="res://scenes/main_menu.tscn"
[autoload]
GameManager="*res://scripts/game_manager.gd"
LoopTracker="*res://scripts/loop_tracker.gd"
[input]
move_left={...} move_right={...} jump={...}
dash={...} interact={...}
```

Turn 5 [user]

```
Now generate the file: scenes/main_menu.tscn
```

Turn 6 [assistant] — main_menu.tscn (86 lines)

```
[gd_scene load_steps=3 format=3]
[ext_resource type="Script" path="res://scripts/main_menu.gd"]
[ext_resource type="Texture2D" path="res://textures/title_bg.png"]
# @asset: background, dark pixel art loop motif, 1280x720, retro style
[node name="MainMenu" type="Control"]
[node name="PlayButton" type="Button" parent="."]
text = "Start Game" ...
```

Turn 7 [user]

```
Now generate the file: scenes/player.tscn
```

Turn 8 [assistant] — player.tscn (312 lines)

```
[gd_scene load_steps=48 format=3]
[ext_resource type="Script" path="res://scripts/player.gd"]
[ext_resource type="Texture2D" path="res://textures/player_run.png"]
# @asset: spritesheet, pixel character running, 8 frames 32x32, side-scroller
[sub_resource type="CapsuleShape2D" ...]
[node name="Player" type="CharacterBody2D"]
[node name="Sprite" type="AnimatedSprite2D" parent="."]
[node name="DashTimer" type="Timer" parent="."]
wait_time = 0.3
...
```

Turn 9 [user]

Now generate the file: scenes/enemy.tscn

Turn 10 [assistant] — enemy.tscn (198 lines)

```
[gd_scene load_steps=28 format=3]
[ext_resource type="Script" path="res://scripts/enemy.gd"]
[ext_resource type="Texture2D" path="res://textures/enemy_walk.png"]
# @asset: spritesheet, skeleton enemy patrol, 6 frames 32x32, pixel art
[sub_resource type="RectangleShape2D" ...]
[node name="Enemy" type="CharacterBody2D"]
[node name="Sprite" type="AnimatedSprite2D" parent="."]
[node name="DetectionZone" type="Area2D" parent="."]
[node name="PatrolTimer" type="Timer" parent="."]
...
```

Turn 11 [user]

Now generate the file: scenes/collectible.tscn

Turn 12 [assistant] — collectible.tscn (72 lines)

```
[gd_scene load_steps=8 format=3]
[ext_resource type="Script" path="res://scripts/collectible.gd"]
[ext_resource type="Texture2D" path="res://textures/orb_glow.png"]
# @asset: sprite, glowing orb collectible, 16x16, pixel art with glow
[node name="Collectible" type="Area2D"]
[node name="Sprite" type="Sprite2D" parent="."]
[node name="CollisionShape" type="CollisionShape2D" parent="."]
[node name="AnimationPlayer" type="AnimationPlayer" parent="."]
...
```

Turn 13 [user]

Now generate the file: scenes/hud.tscn

Turn 14 [assistant] — hud.tscn (95 lines)

```
[gd_scene load_steps=4 format=3]
[ext_resource type="Script" path="res://scripts/hud.gd"]
[node name="HUD" type="CanvasLayer"]
[node name="LoopCounter" type="Label" parent="."]
text = "Loop: 1"
[node name="CollectibleCount" type="Label" parent="."]
[node name="DashCooldown" type="TextureProgressBar" parent="."]
...
```

Turns 15–20: game_world.tscn (1,420 lines), game_manager.gd (85 lines), loop_tracker.gd (42 lines) — generated in dependency order.

Turn 21 [user]

Now generate the file: scripts/player.gd

Turn 22 [assistant] — player.gd (138 lines)

```
extends CharacterBody2D
const SPEED = 200.0
const JUMP_VELOCITY = -350.0
var can_dash: bool = true
var is_dashing: bool = false

func _physics_process(delta):
    if not is_on_floor():
        velocity.y += gravity * delta
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = JUMP_VELOCITY
    var direction = Input.get_axis("move_left", "move_right")
    velocity.x = direction * SPEED if direction else 0
    move_and_slide()
    ...
```

Turn 23 [user]

Now generate the file: scripts/enemy.gd

Turn 24 [assistant] — enemy.gd (76 lines)

```
extends CharacterBody2D
@export var patrol_speed: float = 60.0
@export var chase_speed: float = 120.0
var player_detected: bool = false

func _physics_process(delta):
    if player_detected and player_ref:
        var dir = sign(player_ref.global_position.x - global_position.x)
        velocity.x = dir * chase_speed
    else:
        velocity.x = direction * patrol_speed
    move_and_slide()
```

Turns 25–32: collectible.gd (28 lines), hud.gd (32 lines), camera.gd (22 lines), main_menu.gd (18 lines). Total: 16 turns, 15 files, ~4,200 lines.

F Evaluation Prompts

F.1 Task 1: From-Scratch Generation

Task 1 uses a multi-round protocol: Round 0 generates a blueprint, Rounds 1–N generate files in dependency order.

System Prompt

You are an expert indie game developer competing in a Game Jam. Your goal is to win first place by creating an innovative and polished game. You must demonstrate both creative game design and solid software engineering.

Technical requirements (Godot 4.x):

- The project must contain a valid `project.godot` with `config_version=5`
- Scene files use `.tscn` format, scripts use `.gd` format (GDScript 2.0)
- A main scene must be configured in `project.godot` under `run/main_scene`
- Do not use any external plugins or addons
- For image, audio, and font assets, use semantic file paths and add a description comment above each reference:
@asset: type, description, suggested_size, style
Example: # @asset: sprite, player character idle pose, 32x64, pixel art
Placeholder asset files will be provided automatically -- just reference the paths correctly.

Round 0 — Blueprint (1a: Theme Only)

The Game Jam theme is: "{theme}"

Design a complete 2D game for this theme and output a project blueprint.

Output ONLY the following JSON (no other text):

```
```json
{
 "game_name": "English game name",
 "game_description": "One sentence describing the gameplay",
 "genre": "game genre",
 "core_mechanics": ["mechanic1", "mechanic2", "..."],
 "files": [
 {
 "path": "file path (e.g., project.godot, scenes/xxx.tscn, scripts/xxx.gd)",
 "type": "config / scene / script",
 "description": "what this file does",
 "depends_on": ["paths of files this depends on"]
 }
],
 "assets_needed": [
 {
 "path": "assets/semantic_filename.png",
 "type": "sprite / audio / font",
 "description": "what this asset looks like or sounds like"
 }
]
}
```
```

Rules:

- The first file must be `project.godot`
- Files must be ordered by dependency (dependencies first)
- Design freely -- choose your own structure, number of files, and scope

[Task 1b additionally prepends:]

Gameplay description:
{description}

[And adds the rule:]

- All features mentioned in the gameplay description must be reflected in the file plan

Round 1–N — File Generation (repeated per file)

Now generate the file: {file_path}
Description: {file_description}
Type: {file_type}

Project blueprint:
{blueprint_summary}

Previously generated files:
{generated_summary}

Output ONLY the complete file content. No markdown code blocks, no explanations.
If it's a .tscn scene file, output the full Godot scene format.
If it's a .gd script file, output complete GDScript code.
If it's project.godot, output the full project configuration.

For asset references:

- Use res:// paths for all assets
- Add an @asset comment above each asset reference describing what the asset should be

F.2 Task 2: Code Completion

Task 2 uses four rounds: project overview, context delivery, missing code identification, and code generation.

System Prompt

You are an expert Godot 4.x game developer. You have been given an existing game project that was developed during a Game Jam. Some of the GDScript (.gd) files in this project are missing or incomplete.

Your goal is to restore the project to a fully functional state by analyzing the existing project structure (scene files, project configuration, and any remaining scripts) and writing the missing code.

Technical requirements:

- All code must be compatible with Godot 4.x (GDScript 2.0 syntax)
- Your code must integrate correctly with the existing .tscn scene files and project.godot configuration
- Do not modify any existing files -- only complete the missing or incomplete ones
- Maintain consistency with the coding style and architecture of any remaining scripts
- For asset references, keep any existing paths unchanged
- Functions marked with 'pass # INCOMPLETE: implementation removed' need their full implementation restored

Round 0 — Project Overview

I'm going to show you a Game Jam project that needs some code restored. Let me first give you an overview of what this game is about.

Game Title: {game_name}
Genre: {genre}

Description:
{game_description}

Core Mechanics: {mechanics}
Controls: {controls}

Here is the complete file tree of the project:
{file_tree}

Take a moment to understand the game's design. In the next message, I will show you all the existing files in the project.

Round 1 — Context Files

Here are all the existing files currently in the project. Some .gd script files may be missing entirely, and some may have incomplete function implementations (marked with 'pass # INCOMPLETE: implementation removed').

Please read through all files carefully to understand the project architecture, scene hierarchy, node types, signal connections, and how scripts are referenced.

{all_files_content}

After reviewing these files, I will ask you to identify what is missing and then restore the missing code.

Round 2 — Identify Missing Code

Based on your review of the project, please identify all files that need to be completed or restored.

Consider:

- Script files (.gd) referenced in .tscn scenes but missing from the project
- Script files that exist but have functions marked with 'pass # INCOMPLETE: implementation removed'
- Any other code that is clearly incomplete

For each file, briefly describe what it should implement based on the project context (scene structure, node types, signal connections, and the game description).

Output as JSON:

```
```json
[
 [{"path": "scripts/player.gd", "description": "Player movement, jumping, and collision handling"}]
]
```
```

Round 3 — Generate Code (Level 2b/2c: Script and Full-script)

Now please generate the complete code for ALL the files you identified. Output each file in the following format:

```
=== FILE: path/to/file.gd ===
(complete file content here)
```

```
=== FILE: path/to/another_file.gd ===
(complete file content here)
```

Generate all {n_files} files listed above. For each file:

- Write the complete, working implementation
- Make sure all signals, node references, and exported variables match the .tscn scene definitions
- Match the coding style of the existing project

Output ALL files now, separated by the === FILE: path === markers.

Round 3 — Generate Code (Level 2a: Function-level)

Now please generate ONLY the function implementations that are marked with 'pass # INCOMPLETE: implementation removed'. Do NOT output entire files -- only the individual functions.

Output each function in the following format:

```
=== FUNC: path/to/file.gd::function_name ===
func function_name(args):
  (your complete implementation here)
```

```
=== FUNC: path/to/another_file.gd::another_func ===
func another_func():
  (your complete implementation here)
```

For each function:

- Write the complete function including the 'func' signature line
- Make sure it integrates correctly with the rest of the file and project
- Match the coding style of the existing project

Output ALL incomplete functions now, separated by the === FUNC: path::name === markers.

F.3 Eval Config Generation

Generated once per project during dataset construction (Section 3.3) and reused for all evaluations.

System Prompt

You are an expert Godot 4.x game developer. Your task is to analyze a game project's structure and identify key nodes, properties, and signals that can be used for automated gameplay testing.

You will receive a compressed manifest of a Godot game project containing:

- Game name and main scene
- Input actions defined in the project
- Script files with their extends type, functions, and signals
- Scene files with their node trees

Based on this information, output a JSON evaluation config that identifies:

1. The player node (the main controllable character/object)
2. Score/points tracking (if any)
3. Health/lives tracking (if any)
4. Key gameplay signals (collisions, pickups, damage, etc.)
5. Win/lose conditions (if identifiable)
6. What behaviors to expect when input is injected

IMPORTANT RULES:

- Only identify nodes/properties you can CONFIDENTLY determine from the code structure

- Use actual node paths from the scene files, not guesses
- If you cannot determine something, set it to null
- All node paths should be relative to the scene root (e.g., "Player", "World/Player", "UI/ScoreLabel")
- For signals, use the actual signal names from the scripts
- Output ONLY valid JSON, no markdown, no explanation, no preamble

User Prompt

Analyze this Godot game project and output an evaluation config JSON.

Project manifest:

```

{
  "manifest_json": {}
}

```

Output the following JSON structure (and NOTHING else):

```

{
  "player_node": {
    "scene": "scene file path where player exists",
    "path": "node path within that scene",
    "type": "node type (e.g. CharacterBody2D)",
    "has_animated_sprite": true/false
  },
  "score_tracking": {
    "node_scene": "scene path",
    "node_path": "path to Label/RichTextLabel showing score",
    "property": "text",
    "related_variable": "variable name in script if found"
  },
  "health_tracking": {
    "node_scene": "scene path",
    "node_path": "path to health display node",
    "property": "text or value",
    "related_variable": "variable name"
  },
  "key_signals": [
    {
      "source_scene": "scene path",
      "source_node": "node path",
      "signal_name": "signal name",
      "meaning": "what this signal represents (coin_collected, enemy_killed, damage_taken, etc.)"
    }
  ],
  "gameplay_scenes": ["list of scene paths that contain actual gameplay (not menus)"],
  "menu_scenes": ["list of scene paths that are menus/UI"],
  "expected_behaviors": [
    {
      "input": "action name",
      "expected_effect": "what should happen (player_moves_right, player_jumps, etc.)"
    }
  ],
  "game_type": "platformer/shooter/puzzle/rpg/strategy/card/other",
  "notes": "any additional observations about the game structure"
}

```

Set any field to null if you cannot confidently determine it from the manifest.