V-GameGym: Visual Game Generation for Code Large Language Models

Wei Zhang[1], **Jian Yang**[1], **Renshuai Tao**[3], **Lingzheng Chai, Shawn Guo, Jiajun Wu, Xiaoming Chen**[4], **Ganqu Cui**[1*], **Ning Ding**[1], **Xander Xu**[2], **Hu Wei**[2*], **Bowen Zhou**[1*]
[1]Shanghai AI Lab; [2]Alibaba Group; [3]Beijing Jiaotong University; [4]AIStrong;
zwpride@gmail.com

Abstract

Code large language models have demonstrated remarkable capabilities in programming tasks, yet current benchmarks primarily focus on single modality rather than visual game development. Most existing code-related benchmarks evaluate syntax correctness and execution accuracy, overlooking critical game-specific metrics such as playability, visual aesthetics, and user engagement that are essential for real-world deployment. To address the gap between current LLM capabilities in algorithmic problem-solving and competitive programming versus the comprehensive requirements of practical game development, we present **V-GameGym**, a comprehensive benchmark comprising 2,219 high-quality samples across 100 thematic clusters derived from real-world repositories, adopting a novel clustering-based curation methodology to ensure both diversity and structural completeness. Further, we introduce a multi-modal evaluation framework with an automated LLM-driven pipeline for visual code synthesis using complete UI sandbox environments. Our extensive analysis reveals that V-GameGym effectively bridges the gap between code generation accuracy and practical game development workflows, providing quantifiable quality metrics for visual programming and interactive element generation.

1 Introduction

Recent advances in code large language models (code LLMs) have demonstrated remarkable capabilities in programming tasks, building upon foundational models such as Qwen-Coder (Hui et al., 2024), StarCoder (Li et al., 2023; Lozhkov et al., 2024b), and DeepSeek-Coder (Guo et al., 2024b), establishing strong baselines for code generation (Chen et al., 2021; Zhuo et al., 2024; Liu et al., 2024b), completion (Yang et al., 2024b), and understanding tasks (Lu et al., 2021). These LLMs adopt specialized training strategies combining pre-training on large code corpora from repositories like GitHub, followed by post-training to align outputs with programming best practices.

Figure 1: A visual programming about the flappy bird style arcade game.

The recent LLMs like Claude 4 and GLM-4.5 (Anthropic, 2025; Zeng et al., 2025) exhibit enhanced reasoning capabilities for complex programming scenarios. Further, Kimi-K2 (Team et al., 2025) focuses on long-context code comprehension and generation. *The focus of these advanced LLMs is not on solving algorithmic problems, but rather on visual programming to provide more intuitive demonstrations of model performance.* The open-source community (Chen et al., 2023) has begun developing specialized evaluations for game generation tasks. Visual game synthesis (Tong et al., 2025) further advances this domain by incorporating multi-modal understanding to generate games with coherent visual and interactive elements. However, these approaches primarily focus on code generation accuracy and syntax correctness, overlooking critical game-specific evaluation metrics such as playability, visual aesthetics, user engagement, and performance optimization. The absence of comprehensive evaluation frameworks and targeted improvement methodologies limits the practical deployment of code LLMs in professional game development workflows.

* Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 5613–5641

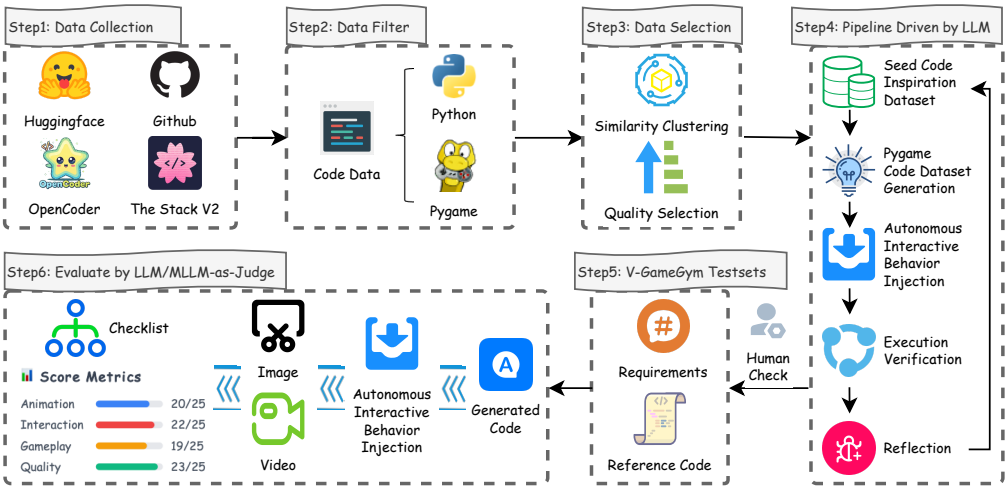


Figure 2: Overview of the V-GameGym framework from data collection to evaluation.

In this work, we first introduce a comprehensive benchmark, **V-GameGym**, comprising 2,219 high-quality samples across 100 thematic clusters derived from real-world Pygame repositories. The process begins by filtering Python source files from large open-source repositories (OpenCoder and The Stack v2) to identify Pygame-related projects, then applies a clustering-based curation strategy that partitions the code corpus using high-dimensional feature vectors and selects the highest-quality program from each cluster based on structural completeness metrics. The curated seed dataset is then processed through an automated LLM-driven pipeline that analyzes code intent, transforms interactive programs into self-contained demonstrations, verifies execution in sandboxed environments with automated error correction, and generates natural language requirement specifications. Finally, the dataset undergoes human validation by 8 graduate students who manually check approximately 2,219 Pygame programs using a complete UI sandbox environment to ensure code integrity and quality.

The contributions are summarized as follows: **(1)** We propose V-GameGym comprised of 2,219 manually verified samples sourced from 2,190 distinct repositories, a comprehensive code generation benchmark for evaluating multimodal game development capabilities, encompassing 100 clusters with diverse functional characteristics. **(2)** We introduce a novel clustering-based curation methodology that combines high-dimensional feature extraction with quality-based selection, ensuring both diversity and structural completeness in the dataset. **(3)** We systematically construct a multimodal evaluation framework with an automated LLM-driven pipeline for code transformation and requirement synthesis, validated through comprehensive human annotation involving 8 graduate students. Notably, extensive analysis reveals that V-GameGym effectively captures the complexity spectrum of real-world game development tasks with quantifiable quality metrics.

2 V-GameGym

2.1 V-GameGym Task Definition

Let $\mathbb{I}$ and $\mathbb{C}$ denote the spaces of natural language instructions and program source codes, respectively. The model under evaluation, $\mathcal{M}_\theta$, is a generative model parameterized by θ that approximates the conditional probability distribution $P(\mathcal{C}|\mathcal{I})$ where $\mathcal{I} \in \mathbb{I}$ and $\mathcal{C} \in \mathbb{C}$. The comprehensive process of generation and evaluation for a given instruction $\mathcal{I}$ is defined by the following sequence.

Code Generation A code instance $\mathcal{C}$ is sampled from the model's output distribution: $\mathcal{C} \sim P_\theta(\cdot|\mathcal{I})$.

Execution & Artifact Synthesis The generated code $\mathcal{C}$ is executed by a deterministic environment function $\mathcal{E} : \mathbb{C} \to \mathbb{A}$, which synthesizes a set of multimedia artifacts $(\mathcal{V}, \mathcal{S}) \in \mathbb{A}$. Here, $\mathbb{A} = \mathbb{V} \times \mathbb{S}$ represents the artifact space, composed of the video space $\mathbb{V}$ and the image space $\mathbb{S}$: $(\mathcal{V}, \mathcal{S}) = \mathcal{E}(\mathcal{C})$.

Multimodal Scoring The quality of the generation is quantified by a comprehensive scoring function by aggregating scores from multiple assessment modalities:

$$\text{Score}(\mathcal{I}, \mathcal{C}, \mathcal{V}, \mathcal{S}) = \sum_{1 \le k \le 3} w_k \cdot \phi_k \quad (1)$$

where $\phi_1(\mathcal{I}, \mathcal{C})$, $\phi_2(\mathcal{I}, \mathcal{S})$, and $\phi_3(\mathcal{I}, \mathcal{V})$ represent modality-specific assessment functions for code,

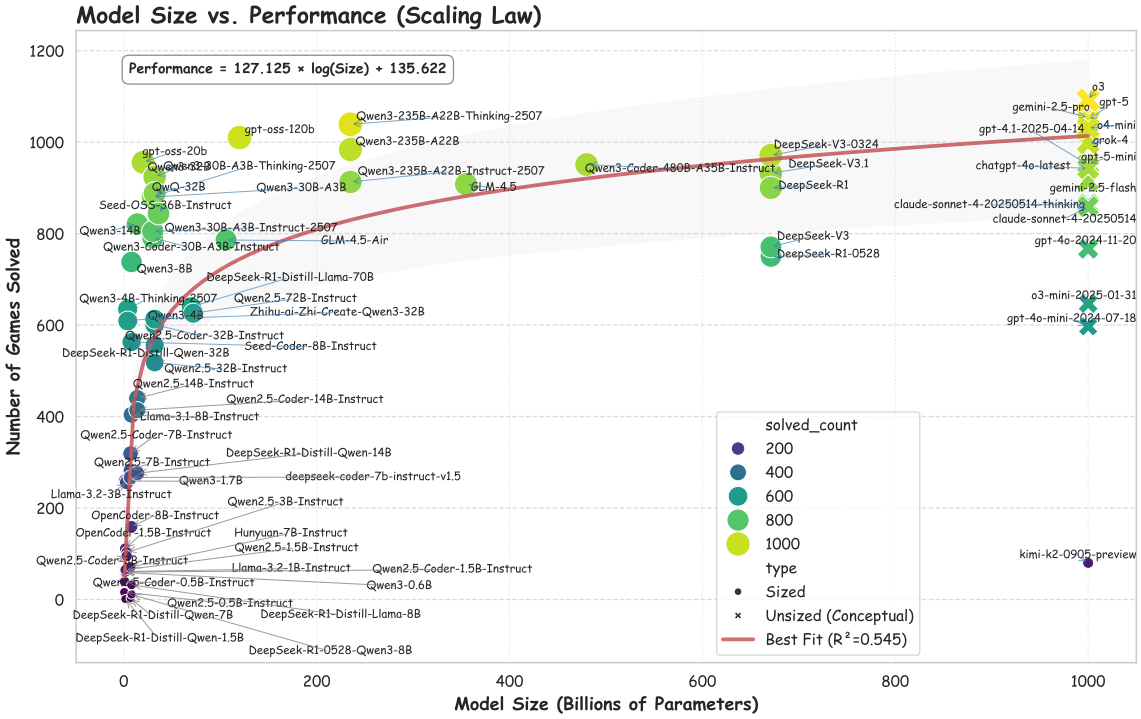


Figure 3: Correlation between model size and games solved.

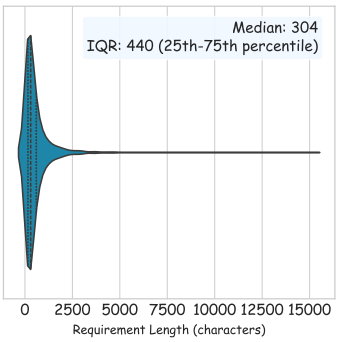


Figure 4: Overall Requirement Length Distribution

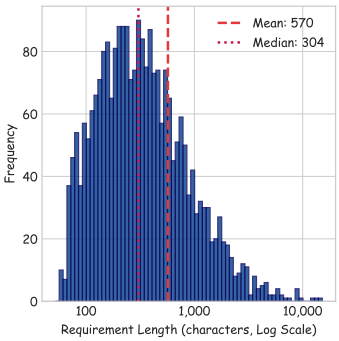


Figure 6: Log-Scale Requirement Length Histogram

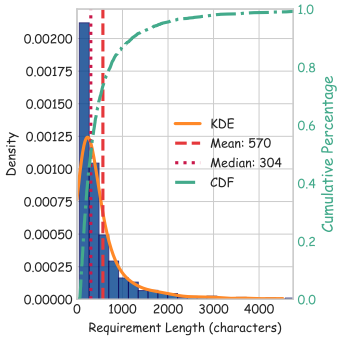


Figure 5: Linear-Scale Requirement Length Histogram

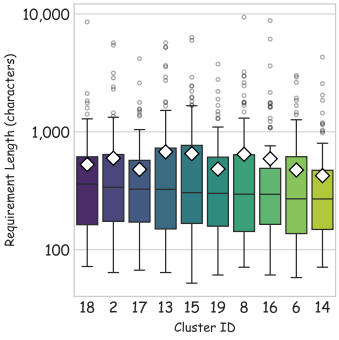


Figure 7: Requirement Length Comparison by Cluster

static visuals, and dynamic gameplay, respectively, and w_k are their corresponding weights satisfying $\sum_k w_k = 1$.

Score Distribution Metrics To provide granular insights into model performance patterns, we categorize each game's final score into four quality bands. **Excellent (80-100)**: Games demonstrating superior implementation quality with minimal issues. **Good (60-80)**: Games with solid functionality and minor deficiencies. **Fair (40-60)**: Games with basic functionality but notable limitations. **Poor (0-40)**: Games with significant implementation failures or non-functional code.

2.2 V-GameGym Construction

Data Collection Our raw data is sourced from two extensive, publicly available code corpora: OpenCoder (Huang et al., 2025) and The Stack v2 (Lozhkov et al., 2024a). To construct a domain-specific dataset, we engineered a high-throughput filtering pipeline. This pipeline leverages parallel processing to stream and analyze all Python source files, systematically isolating code that explicitly contains the "pygame" keyword. This procedure allowed us to efficiently distill a targeted corpus of Pygame-related projects from the broader, more generalized repositories, forming the foundation for all subsequent curation steps.

Clustering-based Curation To ensure the resulting dataset exhibits both high quality and functional diversity, we implemented a rigorous curation strategy that can be described as a formalized selection principle. The process first partitions the entire corpus based on a high-dimensional feature representation and then selects the most structurally complete program from each partition.

Let $D = \{c_1, c_2, \ldots, c_n\}$ be the initial corpus of code samples. Let $\boldsymbol{v}(c) \in \mathbb{R}^d$ be the high-dimensional feature vector extraction function described previously, which maps a code sample c to its quantitative fingerprint (encompassing size, structure, API usage, and semantics). Let $S_{\text{quality}}(c)$ be the scalar heuristic score that evaluates the structural completeness of a program.

The corpus D is first partitioned into k clusters, $C = \{C_1, C_2, \ldots, C_k\}$, using the `MiniBatchKMeans` algorithm on the feature vectors $\boldsymbol{v}(c)$. The final curated seed dataset, D_{seed}, is then constructed by selecting the single element from each cluster that maximizes the quality score S_{quality}:

$$D_{\text{seed}} = \bigcup_{i=1}^{k} \left\{ \arg\max_{c \in C_i} S_{\text{quality}}(c) \right\} \tag{2}$$

where the clusters C_i are the result of $\texttt{MiniBatchKMeans}(D, \boldsymbol{v}, k)$.

This selection principle, articulated in Equation 2, formally captures our two-stage methodology. The clustering operation partitions the dataset based on functional and structural similarity (as defined by $\boldsymbol{v}$), thereby ensuring diversity. The subsequent $\arg\max$ operation within each disjoint set C_i guarantees that the selected program is the most complete and runnable exemplar of that particular functional group, based on the quality heuristic:

$$S_{\text{quality}}(c) = \sum_{f \in C_{\text{struct}}} w_f \cdot \mathbb{I}(f \in c) + S_{\text{len}}(L(c))$$

This decoupling of the clustering metric from the selection metric is intentional. It allows us to group programs by a rich definition of functional behavior ($\boldsymbol{v}$) while applying a simpler, targeted heuristic (S_{quality}) to ensure each chosen sample meets a minimum standard of structural integrity.

Category	Metric	Value
General Statistics		
	Total Samples (Unique Games)	2,219
	Unique Source Repositories	2,190
	Unique Clusters	100
Requirement Metrics (Average per game)		
	Requirement Length (characters)	1,210
	Word Count	178
	Number of Sentences	9.6
Reference Code Metrics (Average per game)		
	Lines of Code	257
	Code Length (characters)	8,533
	Number of Functions	2.8
	Number of Classes	2.4
Execution & Quality Metrics		
	Execution Success Rate	100.0%
	Video Coverage	100.0%
	Average Images per Game	9.9
	Average Recording Duration	10.0 s

Table 1: Overall Statistics of the V-GameGym Dataset.

Test Set Construction The seed dataset is then processed through an automated Language Model (Claude-Sonnet-4)-driven pipeline to construct the final test set, composed of (requirement, code) instruction pairs. This pipeline operationalizes a closed-loop "analyze-inject-validate-generate" workflow. The process commences with an **Intent Analysis** stage, where the LLM parses the seed code to infer its core game mechanics and objectives. This is followed by a **Autonomous Interactive Behavior Injection** stage, which refactors the original, often interactive, code into a self-contained, autonomous demonstration that executes for a fixed duration. The transformed artifact then undergoes **Execution Verification** within a sandboxed environment. Any execution failures initiate a **Self-Correction** loop, wherein the error logs are fed back to the LLM for automated debugging and regeneration. Upon successful validation, **Requirement Generation** module prompts the LLM to synthesize a high-level, natural language requirement specification for the program, emulating the perspective of a product manager. This rigorous process ensures that every entry in the final test

Model	Size	Final	Code	Image	Video	Excellent	Good	Fair	Poor
Proprietary LLMs									
gpt-5	🔒	**45.0**	96.6	17.6	20.7	**83**	288	676	1172
o3	🔒	44.8	92.3	**20.2**	21.9	65	**341**	**686**	1127
gpt-5-mini	🔒	43.5	**96.7**	15.7	18.0	61	236	655	1267
gemini-2.5-pro	🔒	43.5	89.1	19.1	22.2	45	337	672	1165
o4-mini	🔒	43.0	87.8	19.8	21.4	36	313	682	1188
gpt-4.1-2025-04-14	🔒	42.5	91.8	17.6	18.1	47	263	641	1268
grok-4	🔒	42.0	83.9	19.8	**22.4**	21	327	650	1221
gemini-2.5-flash	🔒	42.0	92.8	16.5	16.7	28	252	634	1304
chatgpt-4o-latest	🔒	41.2	82.5	19.9	21.3	25	305	613	1276
claude-sonnet-4-20250514-thinking	🔒	40.5	90.3	14.4	16.9	36	204	624	1355
claude-sonnet-4-20250514	🔒	40.2	87.7	15.7	17.4	36	207	616	1360
o3-mini-2025-01-31	🔒	38.2	89.3	11.9	13.3	26	204	417	1572
gpt-4o-mini-2024-07-18	🔒	33.9	70.4	15.5	15.8	4	134	459	**1622**
400B+ Open-Weight LLMs									
Qwen3-Coder-480B-A35B-Instruct	32B/480B	**41.3**	85.3	18.3	20.5	20	287	**644**	1268
DeepSeek-V3-0324	37B/671B	41.1	83.6	19.3	**20.5**	22	**311**	638	1248
DeepSeek-V3.1	37B/671B	40.9	83.1	**19.3**	20.2	25	296	611	1287
DeepSeek-R1-0528	37B/671B	38.7	**88.1**	13.4	14.6	**32**	174	544	1469
kimi-k2-0905-preview	32B/1000B	23.5	66.3	2.0	2.2	0	18	62	**2135**
100B-400B Open-Weight LLMs									
Qwen3-235B-A22B-Thinking-2507	22B/235B	42.3	84.5	**20.0**	22.4	22	322	**695**	1180
Qwen3-235B-A22B	235B	41.2	81.3	19.8	**22.6**	14	302	668	1235
Qwen3-235B-A22B-Instruct-2507	22B/235B	41.1	85.3	18.2	19.7	16	308	589	1306
GLM-4.5	32B/355B	40.0	84.7	17.0	18.3	31	216	661	1311
GLM-4.5-Air	12B/106B	39.4	85.4	16.3	16.5	23	230	533	**1433**
gpt-oss-120b	5.1B/117B	**43.4**	**90.1**	19.7	20.3	**52**	**324**	634	1209
30B-100B Open-Weight LLMs									
Qwen3-32B	32B	**40.4**	81.6	18.9	20.6	8	274	**642**	1295
Seed-OSS-36B-Instruct	36B	40.3	**88.3**	16.4	16.2	**25**	234	585	1375
Qwen3-30B-A3B-Thinking-2507	3B/30B	40.0	80.7	18.9	20.4	13	**279**	589	1338
QwQ-32B	32B	39.6	79.7	18.5	20.6	10	268	610	1331
Qwen3-30B-A3B	3B/30B	39.6	78.4	**19.7**	**20.7**	9	274	597	1339
Qwen3-Coder-30B-A3B-Instruct	3B/30B	39.0	83.8	16.6	16.7	22	226	543	1428
Qwen3-30B-A3B-Instruct-2507	30B	38.6	81.4	16.5	17.8	11	223	571	1414
DeepSeek-R1-Distill-Llama-70B	70B	35.3	74.1	15.8	16.0	4	188	448	1579
Qwen2.5-72B-Instruct	72B	34.6	73.2	14.7	15.9	3	174	449	1593
Qwen2.5-Coder-32B-Instruct	32B	34.4	74.5	13.8	14.9	9	167	425	1618
DeepSeek-R1-Distill-Qwen-32B	32B	33.4	71.9	14.4	13.9	0	145	411	1663
Qwen2.5-32B-Instruct	32B	31.8	66.4	14.0	15.1	2	127	389	**1701**
10B-30B Open-Weight LLMs									
gpt-oss-20b	3.6B/21B	**42.2**	**88.8**	**18.6**	**19.2**	**31**	**299**	**626**	1263
Qwen3-14B	14B	38.8	79.1	18.4	18.8	9	245	567	1398
Qwen2.5-Coder-14B-Instruct	14B	30.2	68.5	10.9	11.2	0	87	327	1804
DeepSeek-R1-Distill-Qwen-14B	14B	27.4	65.3	8.7	8.3	1	77	198	**1943**
Below 10B Open-Weight LLMs									
Qwen3-8B	8B	**36.9**	**76.2**	**17.2**	**17.3**	**5**	**187**	**546**	1480
Qwen3-4B	4B	34.4	72.7	15.1	15.5	1	144	464	1610
Qwen3-4B-Thinking-2507	4B	34.3	70.0	16.1	16.8	2	168	465	1584
Seed-Coder-8B-Instruct	8B	33.9	73.2	14.0	14.4	4	137	422	1656
Llama-3.1-8B-Instruct	8B	29.4	62.9	13.0	12.3	1	84	319	1815
Qwen2.5-Coder-7B-Instruct	7B	27.6	63.9	9.1	9.7	0	69	250	1899
Qwen3-1.7B	1.7B	25.0	57.3	9.1	8.7	0	43	216	1959
Llama-3.2-3B-Instruct	3B	24.5	55.5	9.5	8.5	0	46	210	1963
deepseek-coder-7b-instruct-v1.5	7B	24.0	53.8	9.0	9.2	0	38	229	1952
Hunyuan-7B-Instruct	7B	21.0	57.8	2.8	2.2	0	12	59	2148
Qwen2.5-Coder-3B-Instruct	3B	20.4	53.8	4.4	2.9	0	10	85	2124
OpenCoder-8B-Instruct	8B	20.1	49.0	6.5	4.7	0	7	152	2060
Llama-3.2-1B-Instruct	1B	15.5	40.3	4.3	1.9	0	4	57	2158
DeepSeek-R1-Distill-Llama-8B	8B	15.1	42.5	1.7	1.0	0	6	26	2187
Qwen3-0.6B	0.6B	13.7	35.1	4.8	1.1	0	3	56	2160
Qwen2.5-Coder-0.5B-Instruct	0.5B	12.8	34.6	3.6	0.0	0	0	39	2179
DeepSeek-R1-Distill-Qwen-7B	7B	12.1	35.8	0.3	0.1	0	0	3	2216
DeepSeek-R1-Distill-Qwen-1.5B	1.5B	8.5	25.4	0.0	0.0	0	0	1	**2218**

Table 2: Comprehensive Performance Evaluation, showing Final Score, Code Score, Image Score, Video Score, and score distribution. **Bold** indicate highest performance; underlined indicate second-highest performance.

set is correct, executable, and paired with a corresponding high-level description.

Human Check and Annotation To ensure the quality of V-GameGym, 8 graduate students used a UI sandbox and LLM assistance to verify nearly 2,219 Pygame code cases and their visual outputs.

2.3 Data Statistics Overview

Data Statistics Table 1 presents a comprehensive statistical overview of the V-GameGym dataset. The benchmark is substantial in scale, comprising 2,219 unique games sourced from 2,190 distinct repositories and organized into 100 thematic clus-

ters. The complexity of the tasks is reflected in the metrics for both the natural language requirements and the reference code; on average, each game's requirements consist of 178 words, while the corresponding reference code implementation spans 257 lines. Critically, the dataset's high quality is underscored by a 100% execution success rate and complete video coverage for all samples, ensuring its reliability for evaluation purposes.

Analysis of V-GameGym Requirements Analysis of the requirement texts reveals a highly right-skewed length distribution, as visualized in the violin plot and histogram (Figures 4, 5). This distribution is characterized by a preponderance of concise specifications, evidenced by a mean length (570) substantially exceeding the median (297), with 80% of texts falling under 1000 characters. On a logarithmic scale, the distribution approximates a log-normal form (Figure 6). Crucially, this length variation correlates with task type. A box plot comparison across the top 10 requirement clusters (Figure 7) demonstrates significant heterogeneity, suggesting that the clustering effectively segments tasks by their underlying complexity or documentation style.

Scaling Law of Visual Game Generation Figure 3, the results reveal a statistically observable positive correlation between the count of model parameters and the performance of the task. The models with smaller parameters (0.5B-3B) exhibit consistently lower performance metrics, typically achieving fewer than 400 solved games, while intermediate-scale models (7B-32B parameters) demonstrate moderate performance ranges of 200-600 solved games. Large-scale models (70B+ parameters) achieve superior performance outcomes, with solved game counts reaching 600-1000+. However, the data suggests that the relationship exhibits logarithmic characteristics rather than linear scaling, indicating diminishing marginal returns as parameter count increases. Significantly, the LLMs (e.g. gpt-oss-20b, Seed-OSS-36B-Instruct, and Qwen3-Coder-30B-A3B) with similar parameters suggest that architectural innovations, training methodologies, and algorithmic optimizations may constitute equally critical factors in achieving state-of-the-art performance. We can obtain the formula for model size and performance as: $M = A * \log(N) + B$, where M is the number of the resolved problems and N is the number of model parameters ($A = 127.2, B = 135.6$).

3 Experiment Setup

Experiment Code LLMs We evaluate all LLMs on Ubuntu 22.04, equipped with an Intel Xeon (R) Gold 6348 CPU @2.60GHz, eight NVIDIA H800 GPUs, and 528 GB of memory. The software setup includes NVIDIA-SMI version 535.104.05 and CUDA 12.3. We set the temperature to 0.0 when inferring by sglang v0.5.1 (Sglang Team).

Evaluated Models For a comprehensive and thorough evaluation, we assess 70 widely used models, including both proprietary and open-source ones. For proprietary models, we evaluate series from leading labs such as OpenAI's GPT (e.g., gpt-5) (OpenAI, 2023, 2025a) and reasoning models (o3, o4-mini) (OpenAI, 2025c), Anthropic's claude-sonnet-4 (Anthropic, 2025), Google's gemini-2.5 series (Google, 2025b,a), and xAI's grok-4 (xAI, 2025). For open-source models, our testing spans a diverse range from major tech companies. This includes the extensive Qwen family (Hui et al., 2024; Qwen, 2025; Yang et al., 2025), ByteDance's Seed series (Seed, 2025; Seed et al., 2025), Moonshot AI's Kimi-K2 (Team et al., 2025), various DeepSeek models (DeepSeek-AI et al., 2025b,a), Meta's Llama series (Llama-3.1, Llama-3.2) (Grattafiori et al., 2024) and Zhipu AI's GLM models (Zeng et al., 2025). The evaluation also incorporates community and research-driven models like OpenAI's gpt-oss (OpenAI, 2025b) and OpenCoder (Huang et al., 2025).

Judge Models We employ an LLM-as-Judge using `Qwen3-Coder-480B-A35B-Instruct` to evaluate code scores and `Qwen2.5-VL-72B` for image/video scores.

4 Analysis

Main Result Note that the sum of distribution counts may not equal 2,219 for some models due to execution failures that prevent complete end-to-end evaluation pipeline completion. In Table 2, several important trends can be observed. **Clear Performance Hierarchy** Proprietary models generally lead, with GPT-5 topping the list at 45.0 points. Among open-source models, large-parameter models (400B+) perform best, such as Qwen3-Coder-480B and the DeepSeek-V3 series, both exceeding 40 points. **Imbalanced Capability Dimensions** All models perform strongly in code generation (most over 70 points) but are generally weaker in image and video evaluation (most under 25 points),

indicating that current models still have significant room for improvement in visual representation and dynamic effect generation. **Pronounced Scale Effect** Open-source models exhibit a clear positive correlation between scale and performance, improving from an average of around 20 points for models under 10B to over 40 points for 400B+ models. **Long-Tail Distribution of Quality** Most generated games fall into the "Poor" and "Fair" categories, with limited samples reaching the "Excellent" standard, reflecting that high-quality game code generation remains challenging.

Multi-Dimensional Capability Analysis of Top-Performing Models Figure 8 highlights distinct model capabilities. A clear code-visual trade-off exists: GPT-5 excels at code (96.6) but is weak in vision (17.6/20.7), while o3 is more balanced and leads in image score (20.2). Notably, open-source models like gpt-oss-120b now rival proprietary systems in game development, narrowing the capability gap.

Evaluation Score Distribution Across Task Dimensions Figure 9 shows a clear hierarchy in AI capabilities for game development. Models excel at code evaluation, achieving high and varied scores, but struggle significantly with visual tasks like screenshot and video evaluations. The consistently low scores in these visual areas highlight a major weakness in the models' visual understanding and generation abilities.

Game Difficulty Distribution Figure 10 presents a typical right-skewed distribution, with most games concentrated in the low solution rate range (where only a few models can solve them), indicating that the games in the test set are generally difficult. The peak on the left shows a considerable number of games that no model could solve, while the number of simple games that could be solved by most models is small. This distribution is beneficial for distinguishing the capability differences among various models.

Performance Analysis Across Game Difficulty Tiers In Figure 11, the difficulty tier analysis reveals that while all models experience performance degradation on harder games, the relative ranking between top-tier models remains remarkably stable across difficulty levels. This consistency validates the benchmark's discriminative power and suggests that superior models maintain their advantages regardless of task complexity.

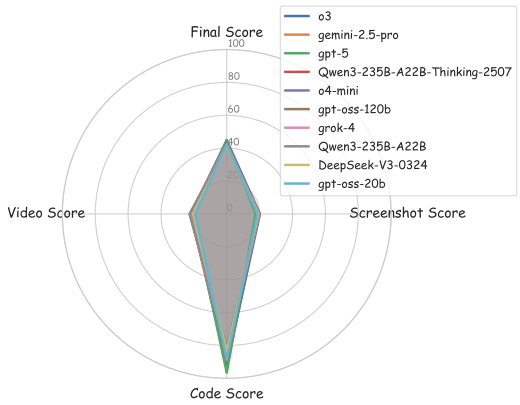


Figure 8: Radar chart comparing the top 10 models across four key performance dimensions.

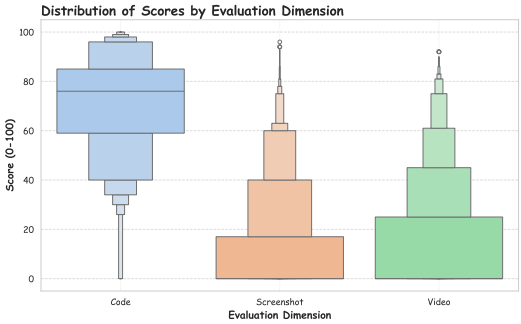


Figure 9: Distribution of evaluation scores across three key dimensions: Code, Screenshot, and Video.

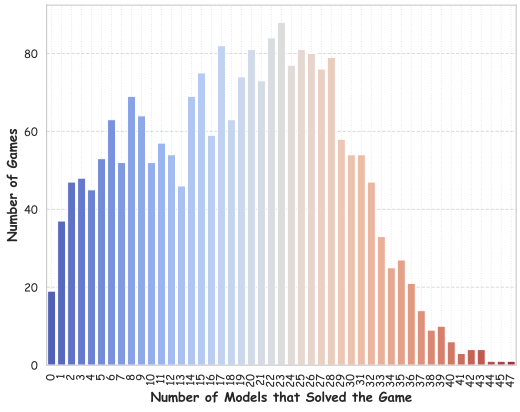


Figure 10: Game difficulty distribution by number of solving models.

Evaluation Dimension Correlations In Figure 12, the correlation analysis shows moderate to strong positive correlations between all evaluation dimensions, indicating that models with superior code generation capabilities tend to also excel in visual assessment tasks. This suggests that game development requires integrated multimodal understanding rather than isolated technical skills.

Overall Correlation vs. Top-Tier Specialization While this positive correlation holds true across the entire model population, a more nuanced picture emerges when examining the elite models, as highlighted in Figure 8. For instance, models

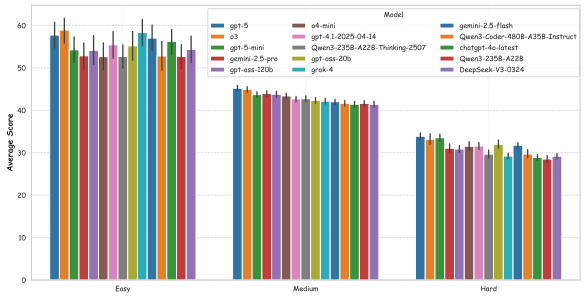


Figure 11: Performance comparison of top 15 models across Easy, Medium, and Hard difficulty tiers, showing consistent ranking patterns and scaling challenges.

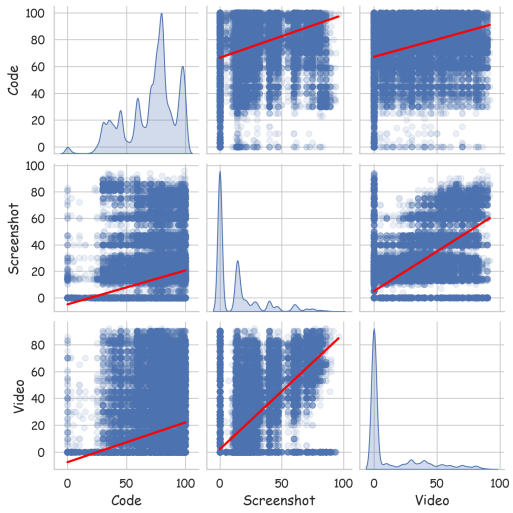


Figure 12: Correlation matrix between Code, Screenshot, and Video evaluation dimensions, demonstrating the interdependence of multimodal capabilities in game development.

like GPT-5 demonstrate a specialization, achieving near-perfect code scores at the expense of comparatively lower visual scores, suggesting a potential "capability trade-off" at the frontier of performance. This indicates that while foundational capabilities are interconnected, advanced models may adopt different strategies to allocate their "reasoning budget", prioritizing either logical code structure or visual aesthetics.

5 Related Work

Code Large Language Models Code-specific large language models (Code LLMs) (Li et al., 2023; Rozière et al., 2023; Guo et al., 2024a; Yang et al., 2024a,b) demonstrate remarkable performance in software engineering and agentic tasks, with foundational models like Qwen2.5/3-Coder (Hui et al., 2024), Seed-Coder (Rozière et al., 2023), GLM-4.5 (Zeng et al., 2025), and Kimi-K2 (Team et al., 2025) excelling in general code generation and understanding. The success of multi-agent collaboration (Guo et al., 2024c; Wang et al., 2023a) inspires the use of a language-specific agent to formulate a multilingual instruction dataset. Subsequently, instruction tuning (Ouyang et al., 2022; Zhang et al., 2023; Wang et al., 2023b) enhances the ability of the LLMs to generalize and follow instructions (Wang et al., 2023b; Chaudhary, 2023; Luo et al., 2023; Wei et al., 2023; Yu et al., 2023). A series of code benchmarks is proposed to evaluate different aspects of the code LLMs, including realistic (Liu et al., 2024b; Zhuo et al., 2024; Zhang et al., 2025b) and multilingual scenarios (Cassano et al., 2023; Chai et al., 2024; Liu et al., 2024a; Zhang et al., 2025a).

Game for Large Language Models The intersection of games and large language models (LLMs) has emerged as a rich area of research encompassing multiple paradigms and applications. Early works established the potential of using game environments as training grounds for LLMs, and then they extended to more complex games (e.g. minecraft (Gong et al., 2024), social deduction games (Light et al., 2023; Wu et al., 2024), text-based adventure games (Guertler et al., 2025)). Subsequent research (Yao et al., 2025) has explored LLMs as players in various game contexts, from traditional board games requiring strategic reasoning to complex multiplayer online environments that demand natural language communication and coordination. The recent work KORGym (Shi et al., 2025) offers over fifty games in either textual or visual formats. But these benchmarks focus on text reasoning, ignoring the evaluation for the code large language model. In this work, we introduce V-GameGym to evaluate the coding capability of LLMs to create the visual games.

6 Conclusion

We introduce V-GameGym, a multimodal benchmark for evaluating code LLMs in visual game generation. Built by curating 2,219 high-quality Pygame samples, our framework assesses both code generation and visual capabilities. Our evaluation of 70 models reveals a significant performance gap between proprietary and open-source models, with top models succeeding only 45%. The benchmark highlights critical limitations in visual understanding and dynamic gameplay generation, providing a foundation for advancing AI-assisted game development.

7 Limitations

We acknowledge the following limitations of our study, **Scope of Language:** The current iteration of V-GameGym focuses exclusively on Python. **Reliance on LLM-based Evaluation:** Our multimodal evaluation pipeline utilizes LLMs as automated judges for assessing code, image, and video quality. While this approach enables scalable and reproducible evaluation, it may not fully capture the nuanced and subjective aspects of game quality, such as playability, aesthetics, and user engagement. **Complexity of Game Projects:** The benchmark primarily consists of single-file, self-contained game implementations.

8 Ethics Statement

This research adheres to ethical guidelines for AI development. We aim to enhance the capabilities of large language models (LLMs) while acknowledging potential risks such as bias, misuse, and privacy concerns. To mitigate these, we advocate for transparency, rigorous bias testing, robust security measures, and human oversight in AI applications. Our goal is to contribute positively to the field and to encourage responsible AI development and deployment.

References

Anthropic. 2025. Claude 3.7 sonnet and claude code.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2023. Multipl-e: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, and 1 others. 2024. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Dake Chen, Hanbin Wang, Yunhao Huo, Yuzhao Li, and Haoyang Zhang. 2023. Gamegpt: Multi-agent collaborative framework for game development. *arXiv preprint arXiv:2310.08067*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. abs/2107.03374.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025b. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Ran Gong, Qiuyuan Huang, Xiaojian Ma, Yusuke Noda, Zane Durante, Zilong Zheng, Demetri Terzopoulos, Li Fei-Fei, Jianfeng Gao, and Hoi Vo. 2024. Mindagent: Emergent gaming interaction. In *Findings of the Association for Computational Linguistics: NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 3154–3183. Association for Computational Linguistics.

Google. 2025a. gemini 2.5-flash.

Google. 2025b. gemini 2.5-pro.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Leon Guertler, Bobby Cheng, Simon Yu, Bo Liu, Leshem Choshen, and Cheston Tan. 2025. Textarena. *arXiv preprint arXiv:2504.11442*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024a. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, and 1 others. 2024b. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024c. Large language model based

multi-agents: A survey of progress and challenges. *CoRR*, abs/2402.01680.

Siming Huang, Tianhao Cheng, J. K. Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2025. Opencoder: The open cookbook for top-tier code large language models. *Preprint*, arXiv:2411.04905.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *CoRR*, abs/2305.06161.

Jonathan Light, Min Cai, Sheng Shen, and Ziniu Hu. 2023. Avalonbench: Evaluating llms playing the game of avalon. *arXiv preprint arXiv:2310.05036*.

Shukai Liu, Linzheng Chai, Jian Yang, Jiajun Shi, He Zhu, Liran Wang, Ke Jin, Wei Zhang, Hualei Zhu, Shuyue Guo, and 1 others. 2024a. Mdeval: Massively multilingual code debugging. *arXiv preprint arXiv:2411.02310*.

Siyao Liu, He Zhu, Jerry Liu, Shulin Xin, Aoyan Li, Rui Long, Li Chen, Jack Yang, Jinxiang Xia, ZY Peng, and 1 others. 2024b. Fullstack bench: Evaluating llms as full stack coder. *arXiv preprint arXiv:2412.00535*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024a. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024b. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

OpenAI. 2025a. Introducing gpt-4.5.

OpenAI. 2025b. Introducing gpt-oss.

OpenAI. 2025c. Introducing o3-and-o4-mini.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Qwen. 2025. Qwq-32b: Embracing the power of reinforcement learning.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

ByteDance Seed. 2025. Seed-oss open-source models release.

ByteDance Seed, Yuyu Zhang, Jing Su, Yifan Sun, Chenguang Xi, Xia Xiao, Shen Zheng, Anxiang Zhang, Kaibo Liu, Daoguang Zan, and 1 others. 2025. Seed-coder: Let the code model curate data for itself. *arXiv preprint arXiv:2506.03524*.

Sglang Team. Sglang project.

Jiajun Shi, Jian Yang, Jiaheng Liu, Xingyuan Bu, Jiangjie Chen, Junting Zhou, Kaijing Ma, Zhoufutu Wen, Bingli Wang, Yancheng He, and 1 others. 2025. Korgym: A dynamic game platform for llm reasoning evaluation. *arXiv preprint arXiv:2505.14552*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Jingqi Tong, Jixin Tang, Hangcheng Li, Yurong Mou, Ming Zhang, Jun Zhao, Yanbo Wen, Fan Song, Jiahao Zhan, Yuyang Lu, Chaoran Tao, and 1 others. 2025. Game-rl: Synthesizing verifiable game tasks at scale to boost vlms general reasoning. *arXiv preprint arXiv:2505.13886*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Ji-Rong Wen. 2023a. A survey on large language model based autonomous agents. *CoRR*, abs/2308.11432.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023b. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 13484–13508. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *CoRR*, abs/2312.02120.

Shuang Wu, Liwen Zhu, Tao Yang, Shiwei Xu, Qiang Fu, Yang Wei, and Haobo Fu. 2024. Enhance reasoning for large language models in the game werewolf. *arXiv preprint arXiv:2402.02330*.

xAI. 2025. grok4.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Jian Yang, Jiaxi Yang, Ke Jin, Yibo Miao, Lei Zhang, Liqun Yang, Zeyu Cui, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2024a. Evaluating and aligning codellms on human preference. *arXiv preprint arXiv:2412.05210*.

Jian Yang, Jiajun Zhang, Jiaxi Yang, Ke Jin, Lei Zhang, Qiyao Peng, Ken Deng, Yibo Miao, Tianyu Liu, Zeyu Cui, and 1 others. 2024b. Execrepobench: Multi-level executable code completion evaluation. *arXiv preprint arXiv:2412.11990*.

Jianzhu Yao, Kevin Wang, Ryan Hsieh, Haisu Zhou, Tianqing Zou, Zerui Cheng, Zhangyang Wang, and Pramod Viswanath. 2025. Spin-bench: How well do llms plan strategically and reason socially? *arXiv preprint arXiv:2503.12349*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *CoRR*, abs/2312.14187.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Renrui Zhang, Jiaming Han, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, Peng Gao, and Yu Qiao. 2023. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. *CoRR*, abs/2303.16199.

Wei Zhang, Jian Yang, Jiaxi Yang, Ya Wang, Zhoujun Li, Zeyu Cui, Binyuan Hui, and Junyang Lin. 2025a. Turning the tide: Repository-based code reflection. *Preprint*, arXiv:2507.09866.

Wei Zhang, Yi Zhang, Li Zhu, Qianghuai Jia, Feijun Jiang, Hongcheng Guo, Zhoujun Li, and Mengping Zhou. 2025b. Adc: Enhancing function calling via adversarial datasets and code line-level feedback. In *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Comprehensive Leaderboard Ranking Models

Figure 13 reveals a clear performance hierarchy with o3 achieving the highest success rate by solving 1,092 games, followed closely by Gemini-2.5-Pro (1,054 games) and GPT-5 (1,047 games). Notably, proprietary models dominate the top positions, with 5 out of the top 6 performers being closed-source systems. Among open-source models, the Qwen3 series demonstrates exceptional performance, with multiple variants appearing in the top rankings. The Qwen3-235B-A22B-Thinking-2507 model achieves the highest open-source performance at 1,039 games solved, ranking 4th overall. The strong showing of thinking-enhanced variants (e.g., Qwen3-235B-A22B-Thinking-2507) suggests that reasoning-augmented architectures provide substantial benefits for complex code generation tasks. The performance gap between the leading models and lower-ranked ones is substantial, with success rates ranging from approximately 49% (1,092/2,219) at the top to 35% (786/2,219) for the 30th-ranked model. This distribution indicates that while current state-of-the-art models can successfully generate functional game code for roughly half of the benchmark tasks, there remains significant room for improvement in achieving consistent, high-quality game development capabilities across diverse requirements.

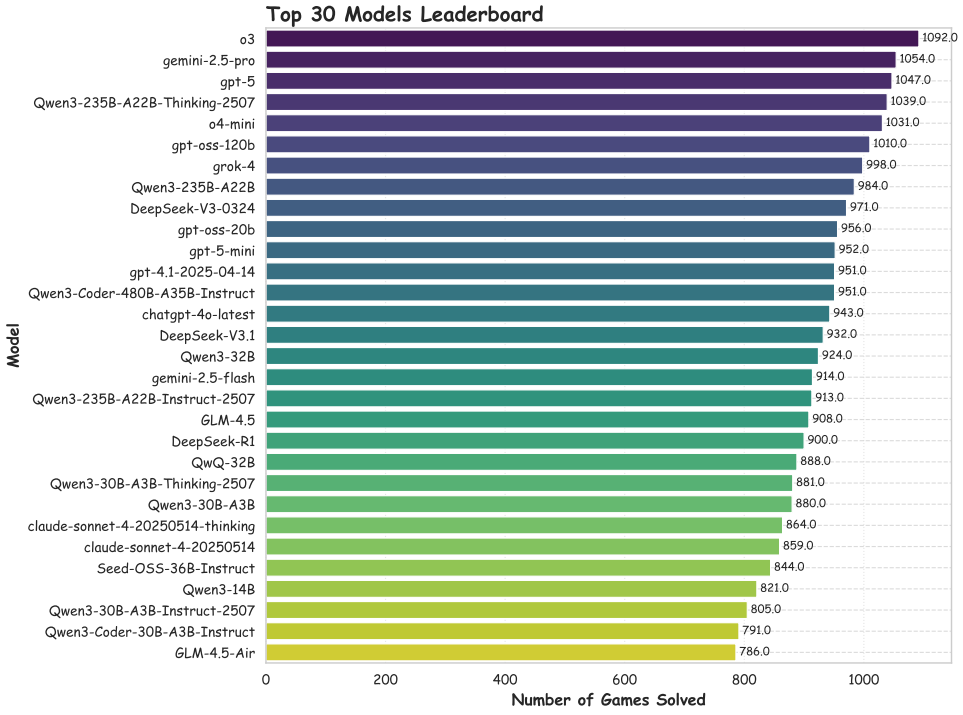


Figure 13: Comprehensive Leaderboard Ranking Models by the Number of Games Successfully Solved Out.

B Complete Leaderboard

Now, we show the complete leaderboard in Table 3.

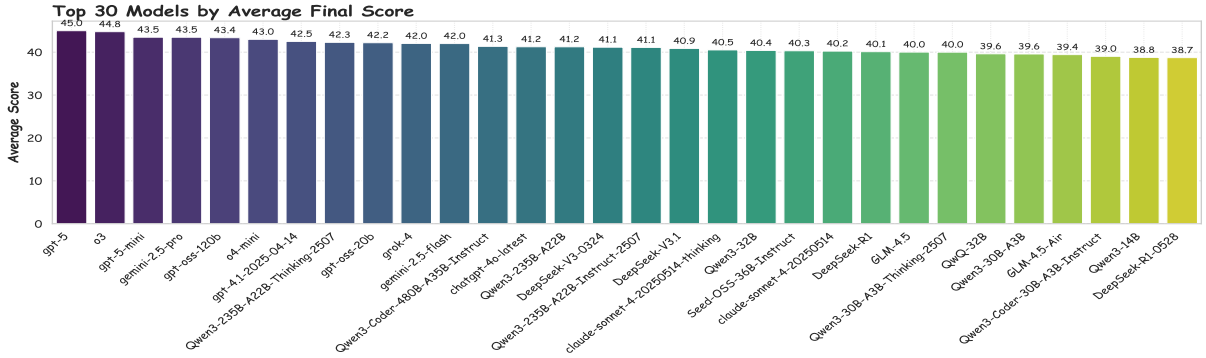
C Comprehensive Performance Comparison Across Different Evaluation Dimensions

Figure 14 presents a comprehensive performance comparison across four key evaluation dimensions. The final performance ranking (a) shows proprietary models dominating the leaderboard, with GPT-5 achieving the highest score of 45.0, followed closely by O3 at 44.8. Code generation performance (b) reveals the strongest capability across all models, with scores ranging from 70-97 points, indicating mature syntactic and logical programming abilities. However, a significant performance gap emerges in visual assessment tasks: image evaluation (c) shows dramatically lower scores (0-20 points), while video evaluation (d) exhibits similar patterns with scores reaching only up to 22.6. This stark contrast between code generation and visual evaluation performance highlights a fundamental challenge in current language

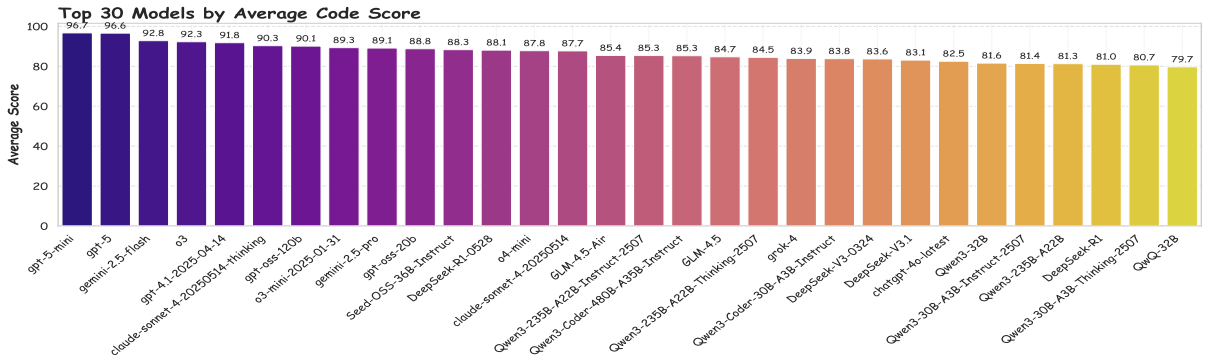
Model	Size	Final	Code	Image	Video	Excellent	Good	Fair	Poor
Proprietary LLMs									
gpt-5	🔒	**45.0**	96.6	17.6	20.7	**83**	288	676	1172
o3	🔒	44.8	92.3	**20.2**	21.9	65	**341**	**686**	1127
gpt-5-mini	🔒	43.5	**96.7**	15.7	18.0	61	236	655	1267
gemini-2.5-pro	🔒	43.5	89.1	19.1	22.2	45	337	672	1165
o4-mini	🔒	43.0	87.8	19.8	21.4	36	313	682	1188
gpt-4.1-2025-04-14	🔒	42.5	91.8	17.6	18.1	47	263	641	1268
grok-4	🔒	42.0	83.9	19.8	**22.4**	21	327	650	1221
gemini-2.5-flash	🔒	42.0	92.8	16.5	16.7	28	252	634	1304
chatgpt-4o-latest	🔒	41.2	82.5	19.9	21.3	25	305	613	1276
claude-sonnet-4-20250514-thinking	🔒	40.5	90.3	14.4	16.9	36	204	624	1355
claude-sonnet-4-20250514	🔒	40.2	87.7	15.7	17.4	36	207	616	1360
o3-mini-2025-01-31	🔒	38.2	89.3	11.9	13.3	26	204	417	1572
gpt-4o-2024-11-20	🔒	37.6	76.6	17.5	18.6	12	224	531	1452
gpt-4o-mini-2024-07-18	🔒	33.9	70.4	15.5	15.8	4	134	459	**1622**
400B+ Open-Weight LLMs									
Qwen3-Coder-480B-A35B-Instruct	32B/480B	**41.3**	85.3	18.3	20.5	20	287	**644**	1268
DeepSeek-V3-0324	37B/671B	41.1	83.6	19.3	**20.5**	22	**311**	638	1248
DeepSeek-V3.1	37B/671B	40.9	83.1	**19.3**	20.2	25	296	611	1287
DeepSeek-R1	37B/671B	40.1	81.0	19.2	20.1	15	278	607	1319
DeepSeek-R1-0528	37B/671B	38.7	**88.1**	13.4	14.6	**32**	174	544	1469
DeepSeek-V3	37B/671B	36.7	73.4	17.7	18.9	3	204	564	1447
kimi-k2-0905-preview	32B/1000B	23.5	66.3	2.0	2.2	0	18	62	**2135**
100B-400B Open-Weight LLMs									
Qwen3-235B-A22B-Thinking-2507	22B/235B	42.3	84.5	**20.0**	22.4	22	322	**695**	1180
Qwen3-235B-A22B	235B	41.2	81.3	19.8	**22.6**	14	302	668	1235
Qwen3-235B-A22B-Instruct-2507	22B/235B	41.1	85.3	18.2	19.7	16	308	589	1306
GLM-4.5	32B/355B	40.0	84.7	17.0	18.3	31	216	661	1311
GLM-4.5-Air	12B/106B	39.4	85.4	16.3	16.5	23	230	533	**1433**
gpt-oss-120b	5.1B/117B	**43.4**	**90.1**	19.7	20.3	**52**	**324**	634	1209
30B-100B Open-Weight LLMs									
Qwen3-32B	32B	**40.4**	81.6	18.9	20.6	8	274	**642**	1295
Seed-OSS-36B-Instruct	36B	40.3	**88.3**	16.4	16.2	**25**	234	585	1375
Qwen3-30B-A3B-Thinking-2507	3B/30B	40.0	80.7	18.9	20.4	13	**279**	589	1338
QwQ-32B	32B	39.6	79.7	18.5	20.6	10	268	610	1331
Qwen3-30B-A3B	3B/30B	39.6	78.4	**19.7**	**20.7**	9	274	597	1339
Qwen3-Coder-30B-A3B-Instruct	3B/30B	39.0	83.8	16.6	16.7	22	226	543	1428
Qwen3-30B-A3B-Instruct-2507	30B	38.6	81.4	16.5	17.8	11	223	571	1414
DeepSeek-R1-Distill-Llama-70B	70B	35.3	74.1	15.8	16.0	4	188	448	1579
Zhihu-ai-Zhi-Create-Qwen3-32B	32B	35.1	75.8	15.2	14.4	3	184	426	1606
Qwen2.5-72B-Instruct	72B	34.6	73.2	14.7	15.9	3	174	449	1593
Qwen2.5-Coder-32B-Instruct	32B	34.4	74.5	13.8	14.9	9	167	425	1618
DeepSeek-R1-Distill-Qwen-32B	32B	33.4	71.9	14.4	13.9	0	145	411	1663
Qwen2.5-32B-Instruct	32B	31.8	66.4	14.0	15.1	2	127	389	**1701**
10B-30B Open-Weight LLMs									
gpt-oss-20b	3.6B/21B	**42.2**	**88.8**	**18.6**	**19.2**	**31**	**299**	**626**	1263
Qwen3-14B	14B	38.8	79.1	18.4	18.8	9	245	567	1398
Qwen2.5-14B-Instruct	14B	30.3	66.4	11.4	13.0	0	92	348	1779
Qwen2.5-Coder-14B-Instruct	14B	30.2	68.5	10.9	11.2	0	87	327	1804
DeepSeek-R1-Distill-Qwen-14B	14B	27.4	65.3	8.7	8.3	1	77	198	**1943**
Below 10B Open-Weight LLMs									
Qwen3-8B	8B	**36.9**	**76.2**	**17.2**	**17.3**	**5**	**187**	**546**	1480
Qwen3-4B	4B	34.4	72.7	15.1	15.5	1	144	464	1610
Qwen3-4B-Thinking-2507	4B	34.3	70.0	16.1	16.8	2	168	465	1584
Seed-Coder-8B-Instruct	8B	33.9	73.2	14.0	14.4	4	137	422	1656
Llama-3.1-8B-Instruct	8B	29.4	62.9	13.0	12.3	1	84	319	1815
Qwen2.5-Coder-7B-Instruct	7B	27.6	63.9	9.1	9.7	0	69	250	1899
Qwen2.5-7B-Instruct	7B	26.1	59.8	9.2	9.2	0	52	230	1937
Qwen3-1.7B	1.7B	25.0	57.3	9.1	8.7	0	43	216	1959
Llama-3.2-3B-Instruct	3B	24.5	55.5	9.5	8.5	0	46	210	1963
deepseek-coder-7b-instruct-v1.5	7B	24.0	53.8	9.0	9.2	0	38	229	1952
Hunyuan-7B-Instruct	7B	21.0	57.8	2.8	2.2	0	12	59	2148
Qwen2.5-Coder-3B-Instruct	3B	20.4	53.8	4.4	2.9	0	10	85	2124
OpenCoder-8B-Instruct	8B	20.1	49.0	6.5	4.7	0	7	152	2060
Qwen2.5-3B-Instruct	3B	18.7	46.9	5.0	4.1	0	9	94	2116
Qwen2.5-Coder-1.5B-Instruct	1.5B	17.3	46.6	3.9	1.2	0	2	58	2159
OpenCoder-1.5B-Instruct	1.5B	16.9	43.1	5.6	1.9	0	9	102	2108
Llama-3.2-1B-Instruct	1B	15.5	40.3	4.3	1.9	0	4	57	2158
Qwen2.5-1.5B-Instruct	1.5B	15.2	40.1	3.7	1.8	0	2	63	2154
DeepSeek-R1-Distill-Llama-8B	8B	15.1	42.5	1.7	1.0	0	6	26	2187
DeepSeek-R1-0528-Qwen3-8B	8B	14.0	41.3	0.4	0.3	0	3	8	2207
Qwen3-0.6B	0.6B	13.7	35.1	4.8	1.1	0	3	56	2160
Qwen2.5-Coder-0.5B-Instruct	0.5B	12.8	34.6	3.6	0.0	0	0	39	2179
DeepSeek-R1-Distill-Qwen-7B	7B	12.1	35.8	0.3	0.1	0	0	3	2216
Qwen2.5-0.5B-Instruct	0.5B	10.9	30.8	1.7	0.0	0	0	16	2200
DeepSeek-R1-Distill-Qwen-1.5B	1.5B	8.5	25.4	0.0	0.0	0	0	1	**2218**

Table 3: Comprehensive Performance Evaluation, showing Final Score, Code Score, Image Score, Video Score, and score distribution. **Bold** indicate highest performance; underlined indicate second-highest performance.

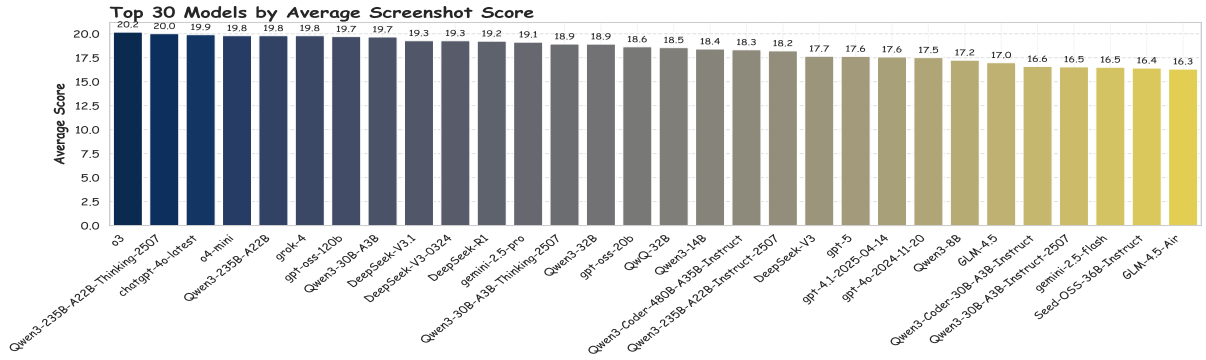
models - while they excel at producing syntactically correct and logically sound code, generating visually appealing and functionally rich interactive games remains substantially more difficult. The consistent ranking patterns across visual modalities suggest that models capable of generating better static visual content also tend to produce superior dynamic gameplay experiences.



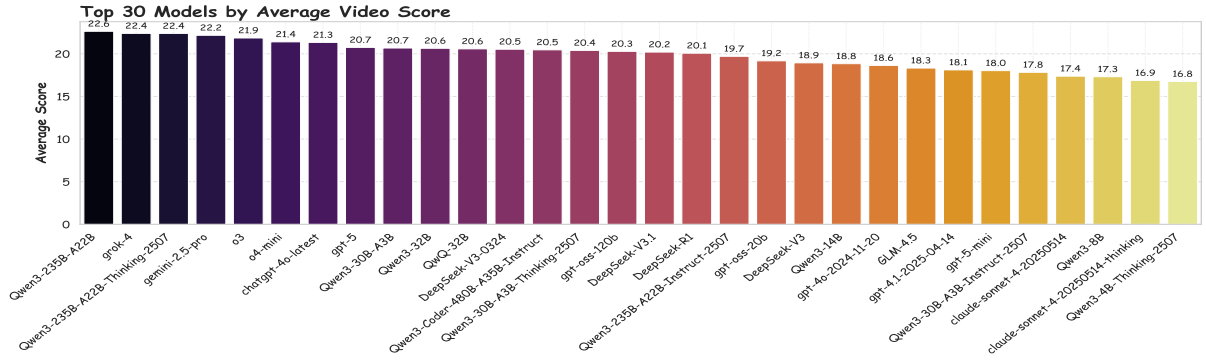
(a) Final Performance



(b) Code Generation Performance



(c) Image Evaluation Performance

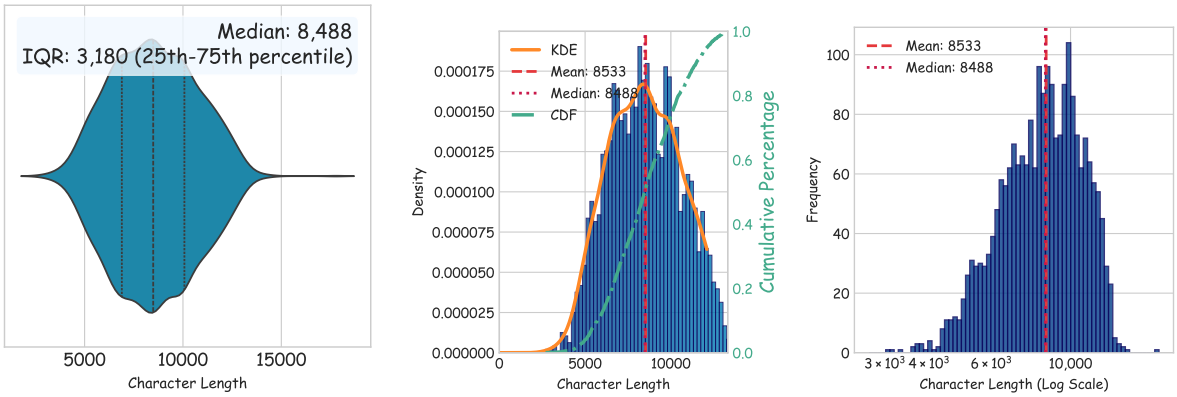


(d) Video Evaluation Performance

Figure 14: Comprehensive performance comparison across different evaluation dimensions. Performance comparison across four evaluation dimensions: (a) Overall scores with GPT-5 and o3 leading, (b) High code generation scores (70-97), (c-d) Low visual evaluation scores (0-20), revealing models excel at code generation but struggle with visual assessment.

D V-GameGym Reference Code Analysis

To comprehensively analyze the character length of reference code within the dataset, we employed three complementary visualization methods. Figure 15(a), a Violin Plot, reveals the overall probability density distribution of the data, exhibiting a clear unimodal shape with its peak concentrated around 8,500 characters. This figure intuitively displays the core statistical characteristics of the data: a median of 8,488 characters, with 50% of the data falling within an interquartile range (IQR) spanning 3,180 characters. Figure 15(b) provides a more refined depiction of this distribution through a histogram with a kernel density estimate (KDE) curve under linear coordinates. The calculated mean of 8,533 characters is notably close to the median, suggesting an approximately symmetrical distribution. Concurrently, the cumulative distribution function (CDF) curve on the right offers a quantitative perspective on the data; for instance, approximately 80% of code samples have a length below 10,000 characters. Finally, to effectively examine the data's full dynamic range, particularly its long-tail portion, Figure 15(c) employs a logarithmic axis. This view compresses the larger value ranges, allowing extreme values at both ends of the distribution to be clearly presented, thus completely illustrating the entire distribution from the shortest to the longest code segments. In summary, these three figures collectively provide a detailed and multifaceted representation of the dataset's central tendency, dispersion, and distributional shape.



(a) Violin plot showing distribution peaked at 8,500 characters.

(b) Histogram showing symmetric distribution.

(c) Log-scale view showing full range distribution.

Figure 15: Comprehensive analysis of reference code character length distribution using three complementary visualization methods: density estimation, linear-scale histogram, and logarithmic-scale representation.

E V-GameGym Word Cloud Analysis

Figure 16 presents a word cloud analysis comparing the linguistic content of the natural language requirements against the Python code solutions in our dataset. The Requirements Word Cloud (left) highlights a strong focus on core game mechanics, with dominant terms such as `player`, `screen`, `game`, and `create`. This confirms the prompts are well aligned with the intended domain. The Code Tokens Word Cloud (right) reveals the most frequent Pygame API calls and programming constructs, including `render`, `font`, `random`, and `time`, outlining the key technical skills required. The clear semantic alignment between the two clouds demonstrates a direct and coherent mapping from the problem descriptions to their programmatic solutions, validating the dataset's suitability for evaluating an LLM's code generation capabilities in this domain.

F V-GameGym Reference Code Patterns Quantitative Analysis

To deeply understand the inherent structure and common practices of code within the V-GameGym dataset, we conducted a quantitative analysis of code patterns, with results shown in Figure 18. The results reveal library usage frequency, core game loop mechanisms, code structure paradigms, and overall complexity distribution, respectively. On one hand, they generally adhere to standard Pygame development paradigms; on the other hand, they exhibit significant diversity in code structure and complexity, making this dataset an ideal resource for training and evaluating code generation models.

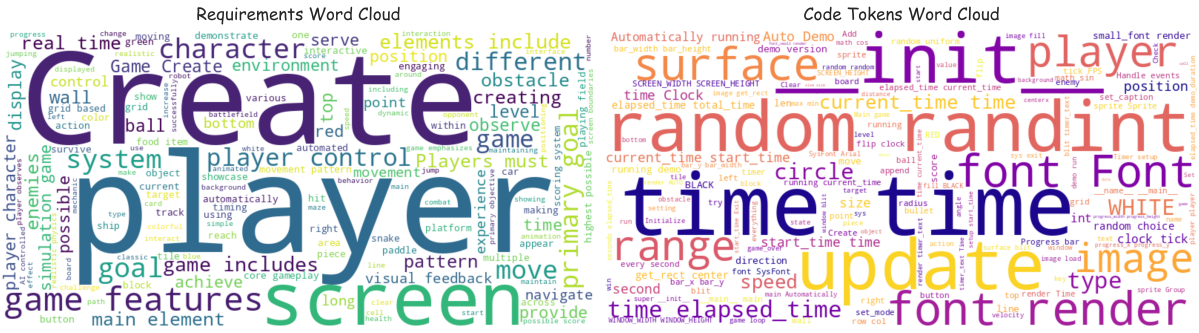


Figure 16: V-GameGym Comparative Word Cloud Analysis of Requirements and Code Corpora.

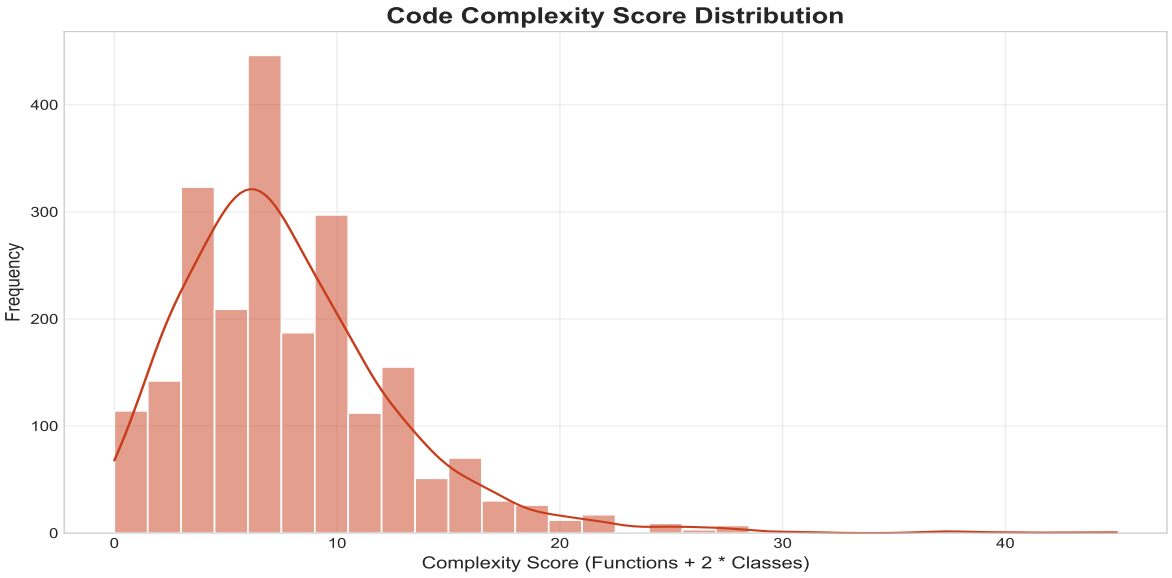
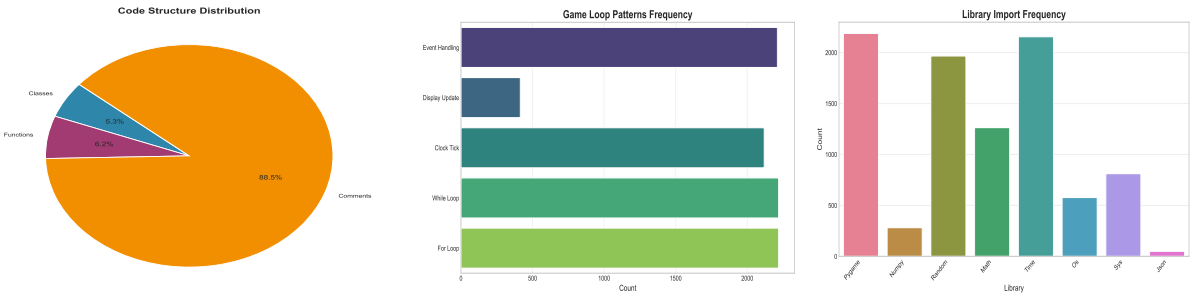


Figure 17: Distribution of code complexity scores.



(a) Occurrence frequency of core game loop patterns.

(b) Distribution of code structure elements.

(c) Import frequency of common Python libraries.

Figure 18: Comprehensive analysis of reference code character length distribution.

Library Import Frequency This plot displays the most frequently imported Python libraries in the V-GameGym dataset. The results indicate that `pygame` is widely used as the core framework, while `random` and `math` libraries also appear frequently, reflecting the common presence of random elements and mathematical computation requirements in game development. This confirms that the samples in the dataset follow standard `Pygame` development practices.

Game Loop Patterns This plot quantifies the occurrences of key code patterns that constitute a typical game loop. The high frequency of calls to `pygame.event` highlights the central role of event-

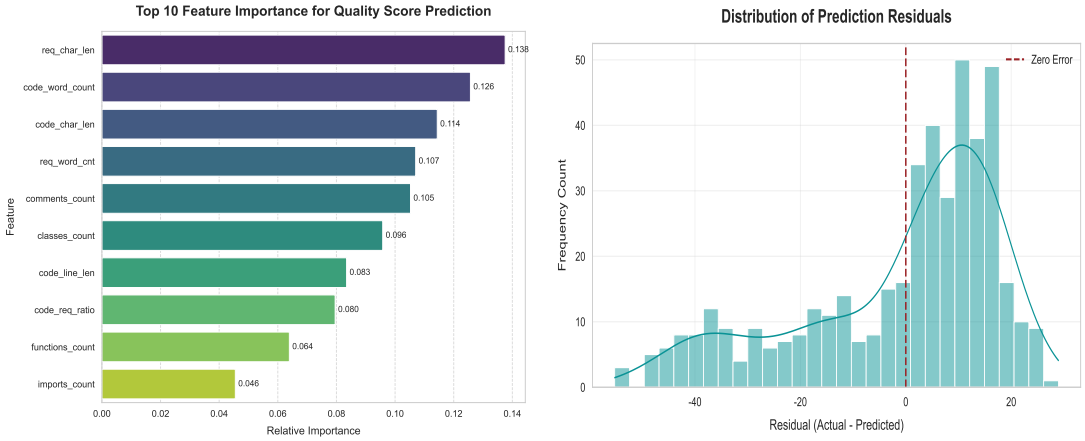
driven programming in game interaction. Similarly, the frequent use of `pygame.display.update` and `clock.tick`, corresponding to screen rendering and frame rate control respectively, is fundamental for building real-time, smooth gaming experiences.

Code Structure Distribution This pie chart depicts the relative proportions of classes, functions, and comments within the code. Analysis shows that functions are the primary units of code organization, while the use of classes also accounts for a significant proportion, indicating a certain application of object-oriented programming (OOP) principles in the samples. The proportion of comments provides an indirect measure of code readability and maintainability.

Code Complexity Score Distribution To assess the structural complexity of the code, we defined a complexity score (calculated as number of functions + 2 * number of classes). The histogram in this figure shows that the complexity scores exhibit a right-skewed distribution, indicating that most game codes in the dataset have relatively simple structures, but it also includes a portion of complex projects with highly intricate structures (e.g., a large number of classes and functions).

G V-GameGym Quality Score Prediction Model Results

To evaluate the performance of our Random Forest regression model for quality score prediction, a multifaceted analysis was conducted, as illustrated in Figure 19.



(a) Comparison of actual vs. predicted quality scores.

(b) Distribution of prediction residuals.

Figure 19: Quality Score Prediction Distribution.

Feature Importance This panel presents the top ten most influential features in determining the model's predictions, ranked by their Gini importance. The analysis reveals that metrics related to code volume and complexity, such as `code_char_len` (total characters in the code) and `code_word_count`, are the strongest predictors. This insight underscores the significant relationship between the sheer size of the codebase and its perceived quality score within this dataset.

Residuals Distribution This histogram displays the distribution of the prediction residuals, calculated as the difference between the actual and predicted scores (Actual - Predicted). The distribution is approximately centered around zero and exhibits a quasi-normal shape, suggesting that the model has no systematic bias (i.e., it does not consistently over- or under-predict). This desirable characteristic indicates that the model's errors are random, which is a key assumption for a well-fitted regression model.

H V-GameGym Distribution of Game Samples Across the Top 30 Source Repositories

Figure 20 provides a quantitative analysis of the contribution frequency from the top 30 source repositories within the curated dataset. The horizontal bar chart illustrates the number of game samples sourced from each unique repository, which are ranked in descending order of their contribution count. A prominent

characteristic revealed by the visualization is the highly granular and flat distribution of samples. The data indicates that the contributions are thinly spread across a wide array of sources, with the most frequent repositories supplying a maximum of only three game samples. A substantial cohort of repositories provided two samples each, followed by another group contributing single instances. This flat, long-tail distribution pattern underscores the extensive diversity of the dataset's origins. By sourcing a small number of games from a large pool of independent repositories, we effectively minimize the risk of stylistic and structural bias that could arise from over-representing a few dominant sources. The resulting heterogeneity ensures a broad and more representative collection of programming patterns, architectural designs, and implementation logic. This characteristic is fundamental to the dataset's objective of serving as a robust foundation for training generalizable models in tasks such as automated code generation and program analysis.

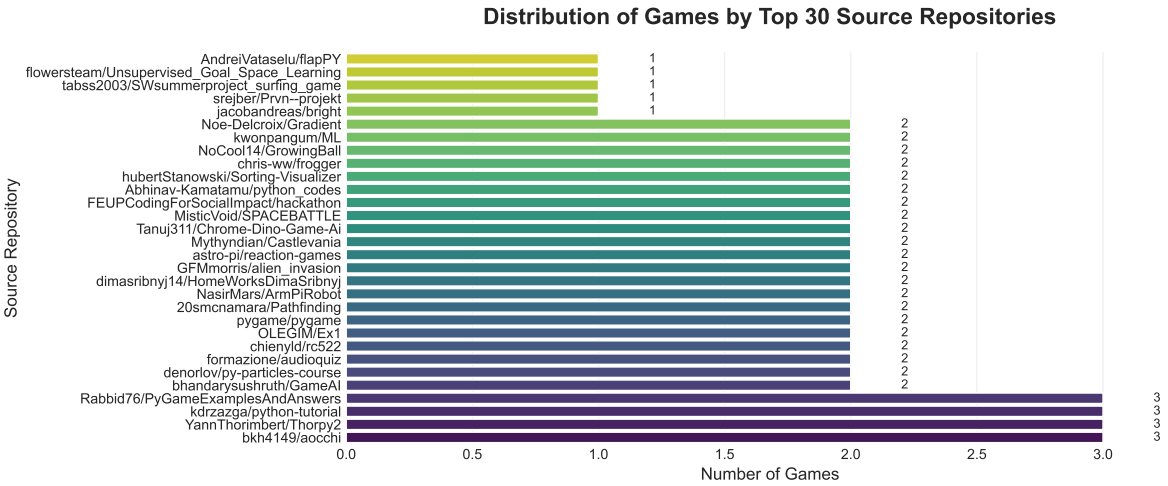


Figure 20: Distribution of game samples across the top 30 source repositories.

I Model Similarity Analysis

In Figure 21, the similarity clustering reveals distinct model families with comparable problem-solving patterns. Models from the same architecture family (e.g., Qwen3 variants, DeepSeek series) tend to cluster together, indicating that foundational architecture and training methodologies significantly influence which games models can successfully solve. Interestingly, some cross-family clusters emerge between models of similar scale, suggesting that parameter count plays a crucial role in determining capability overlap beyond architectural differences.

J Score Threshold Sensitivity Analysis

In Figure 22, the threshold sensitivity analysis demonstrates remarkable ranking stability across different score cutoffs. As the threshold increases from 20 to 80 points, all models show expected performance degradation, but their relative positions remain largely unchanged. This robustness validates our evaluation methodology and suggests that the observed performance differences reflect genuine capability gaps rather than evaluation artifacts. The parallel decline curves indicate that our scoring system maintains discriminative power across the full quality spectrum.

K Score Distribution Characteristics

Figure 23 reveals diverse score distribution patterns among top-performing models. Some models exhibit narrow, concentrated distributions around their median scores, indicating consistent performance across different game types. Others show broader, multi-modal distributions, suggesting specialized strengths in particular game categories. The distribution shapes provide insights into model reliability - models

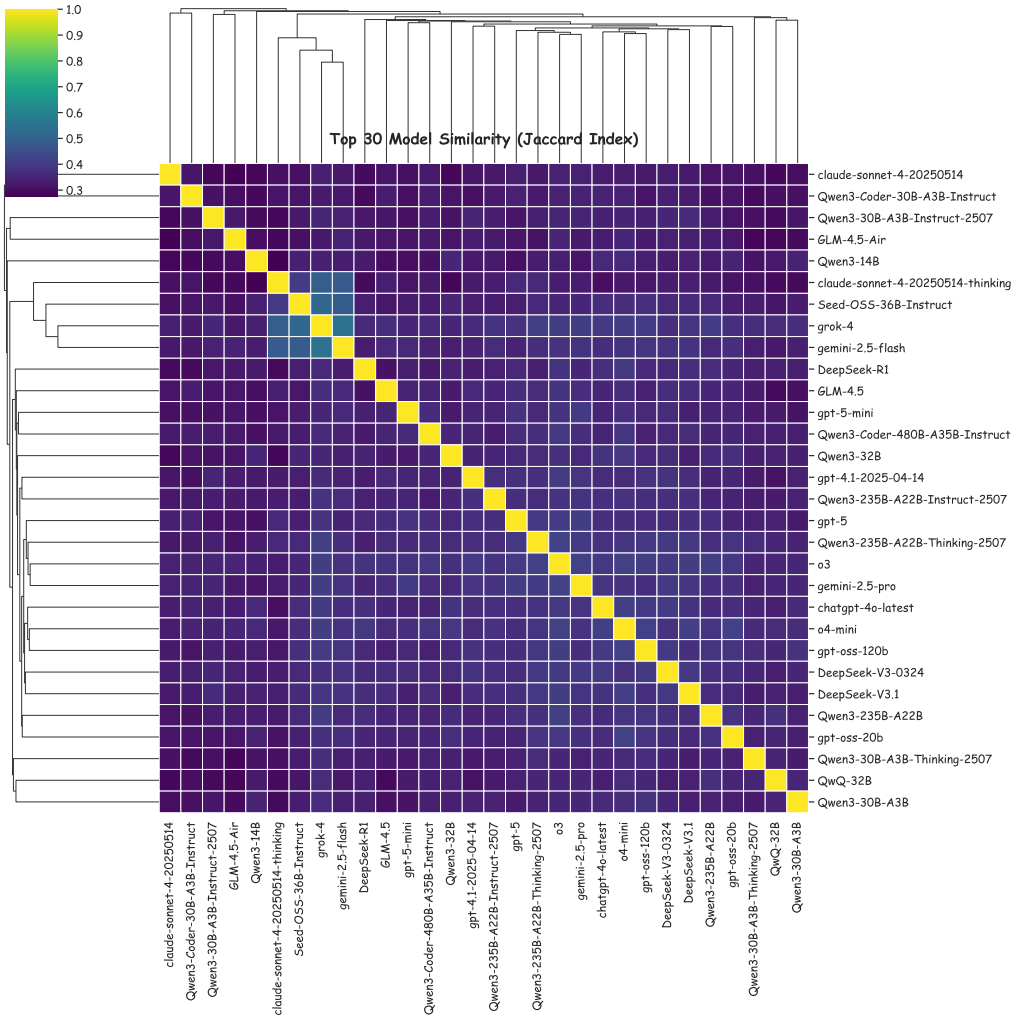


Figure 21: Hierarchical clustering of models based on solved game overlap using Jaccard similarity index.

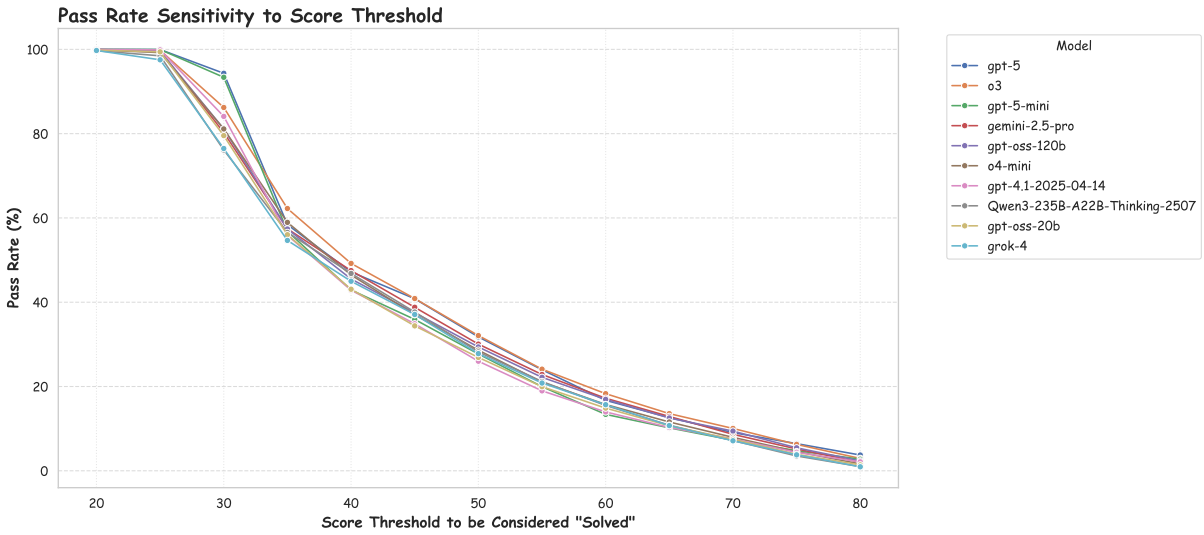


Figure 22: Pass rate variations across different score thresholds, demonstrating ranking stability.

with tighter distributions may be more predictable for production use, while those with wider distributions might excel in specific domains but struggle with others.

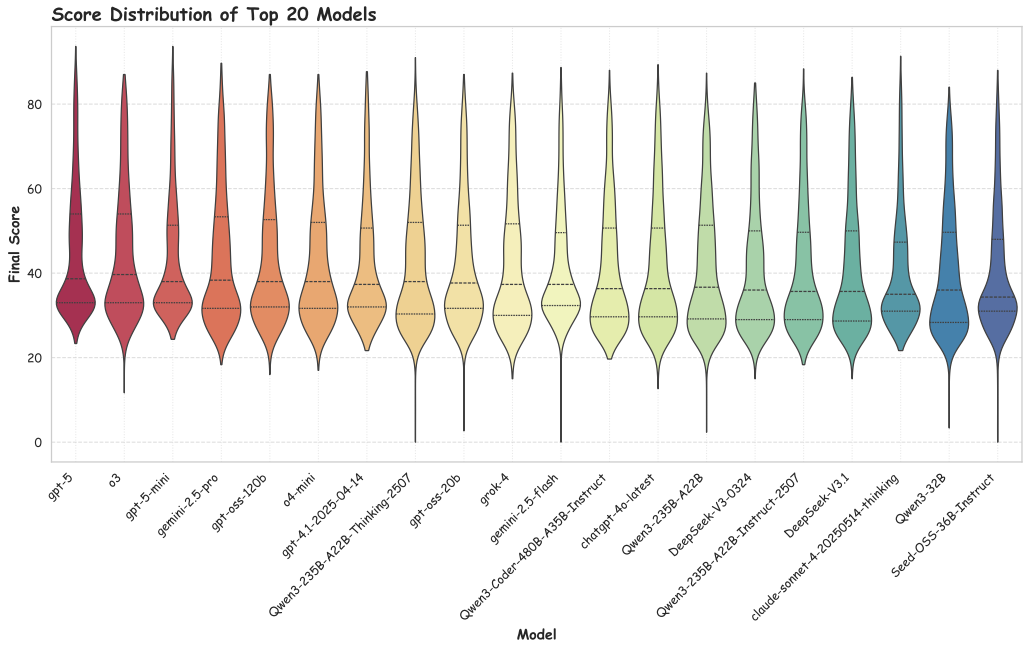


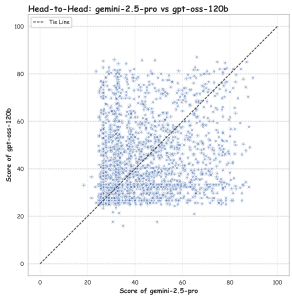
Figure 23: Violin plots showing score distribution patterns for top 20 models.

L Representative Head-to-Head Comparisons

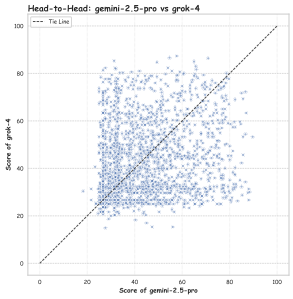
In Figure 24, the head-to-head comparisons reveal nuanced competitive dynamics between leading models. Points above the diagonal line indicate games where the y-axis model outperforms the x-axis model, and vice versa. The scatter patterns show that even among top-tier models, performance advantages are game-specific rather than universal. Some model pairs exhibit complementary strengths, suggesting potential ensemble benefits. The analysis also reveals that certain games consistently favor particular model architectures, indicating systematic biases in problem-solving approaches.

M Comprehensive Performance Heatmap

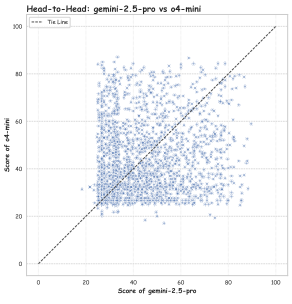
In Figure 25, the performance heatmap provides a granular view of model capabilities across the most challenging benchmark subset. The clear progression from lighter (better performance) to darker (poorer performance) colors as difficulty increases confirms the validity of our difficulty ordering. Notably, even the highest-performing models struggle with the rightmost games, indicating these represent genuine frontier challenges. The heatmap also reveals interesting patterns where certain models show unexpected strength on specific difficult games, suggesting specialized capabilities that average performance metrics might obscure. The clustering of similar performance patterns across model families reinforces the architectural influence on problem-solving approaches.



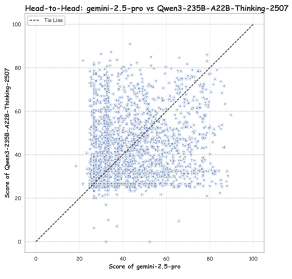
(a) Head-to-head comparison of Gemini-2.5-pro and GPT-OSS-120B.



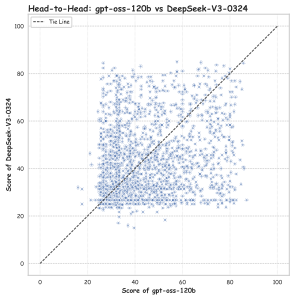
(b) Head-to-head comparison of Gemini-2.5-pro and Grok-4.



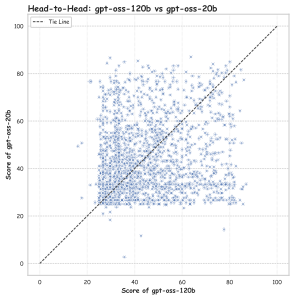
(c) Head-to-head comparison of Gemini-2.5-pro and o4-mini.



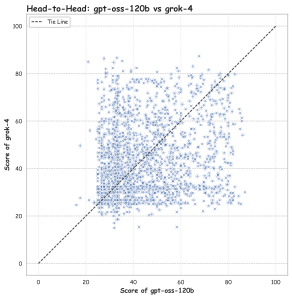
(d) Head-to-head comparison of Gemini-2.5-pro and Qwen3-235B-A22B-Thinking-2507.



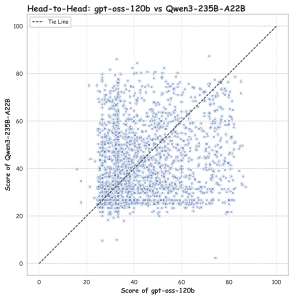
(e) Head-to-head comparison of GPT-OSS-120B and DeepSeek-V3-0324.



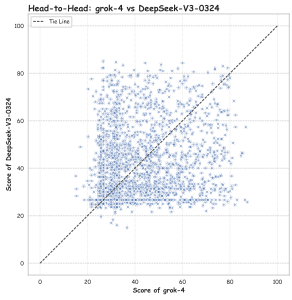
(f) Head-to-head comparison of GPT-OSS-120B and GPT-OSS-20B.



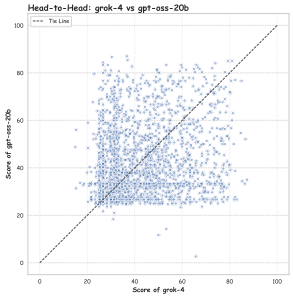
(g) Head-to-head comparison of GPT-OSS-120B and Grok-4.



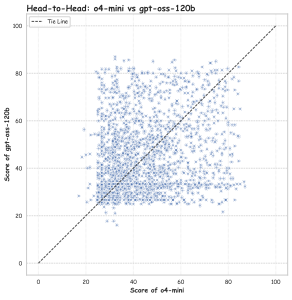
(h) Head-to-head comparison of GPT-OSS-120B and Qwen3-235B-A22B.



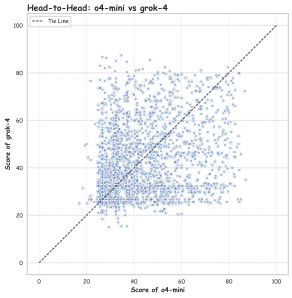
(i) Head-to-head comparison of Grok-4 and DeepSeek-V3-0324.



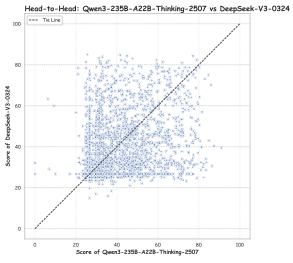
(j) Head-to-head comparison of Grok-4 and GPT-OSS-20B.



(k) Head-to-head comparison of o4-mini and GPT-OSS-120B.



(l) Head-to-head comparison of o4-mini and Grok-4.



(m) Head-to-head comparison of Qwen3-235B-A22B-Thinking-2507 and DeepSeek-V3-0324.

(n) Head-to-head comparison of Qwen3-235B-A22B-Thinking-2507 and GPT-OSS-120B.

(o) Head-to-head comparison of Qwen3-235B-A22B-Thinking-2507 and Grok-4.

(p) Head-to-head comparison of Qwen3-235B-A22B-Thinking-2507 and o4-mini.

Figure 24: Direct performance comparisons between selected model pairs showing competitive advantages across individual games.

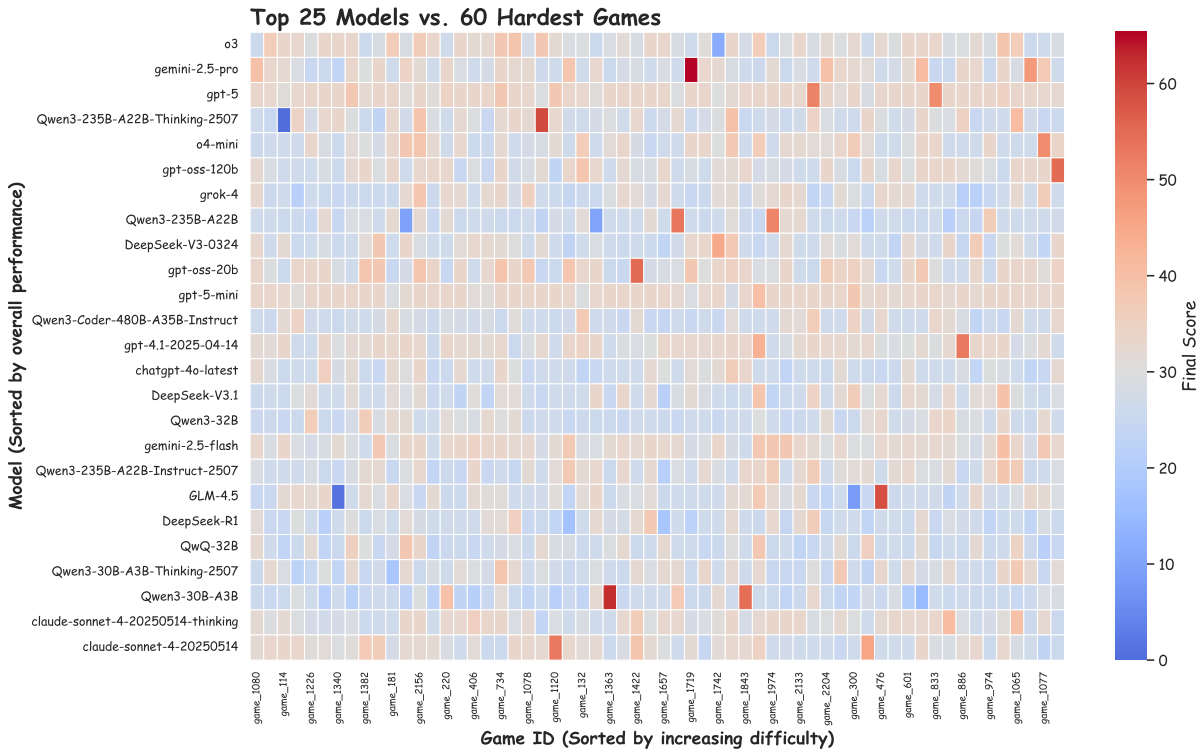


Figure 25: Performance matrix of top 25 models on the 60 most challenging games, ordered by increasing difficulty and overall model performance.

N Seed Code Dataset Quality Analysis

To provide comprehensive insights into the characteristics and quality of our seed dataset, we conducted extensive statistical analysis across multiple dimensions. The analysis encompasses cluster distribution, quality metrics, file characteristics, module usage patterns, and structural complexity.

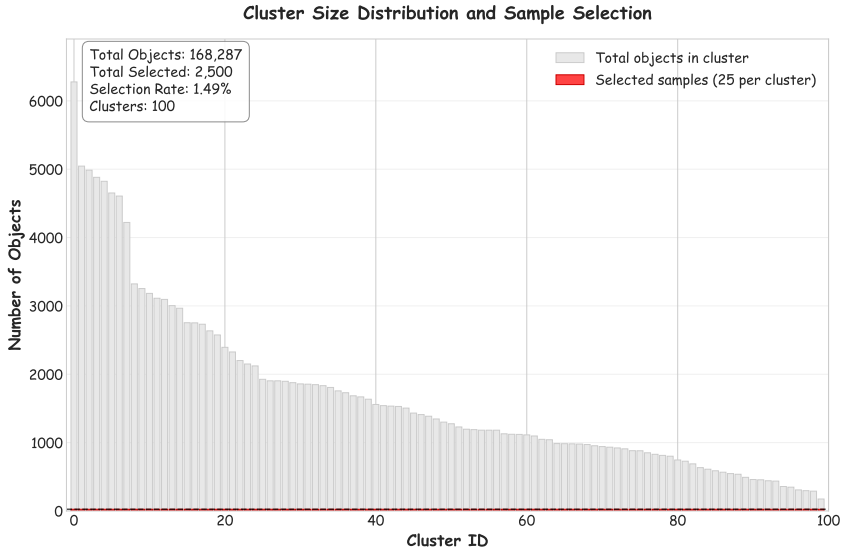


Figure 26: Cluster size distribution and sample selection strategy, showing uniform selection of 25 samples from each of the 100 clusters across 168,287 total objects.

Cluster Coverage and Sample Selection Figure 26 illustrates the distribution of objects across our 100 clusters and the uniform selection strategy employed. The analysis reveals significant variation in cluster sizes, ranging from hundreds to thousands of objects per cluster, with a total of 168,287 objects processed. Our systematic selection of 25 samples per cluster ensures balanced representation across all functional categories, achieving a 1.49% overall selection rate. This uniform sampling strategy effectively mitigates bias toward popular game types while maintaining diversity across the entire functional spectrum.

Quality Score Distribution Analysis Figure 27 demonstrates the high quality of our curated dataset, with 79.7% of samples achieving quality scores above 70. The distribution exhibits a strong right skew with a mean score of 86.2 and a median of 100.0, indicating that our clustering-based selection successfully identified structurally complete and well-implemented code samples. The concentration of samples in the high-quality range validates our selection methodology and ensures that the benchmark provides reliable reference implementations for evaluation purposes.

File Size Characteristics The file size analysis in Figure 28 reveals a log-normal distribution with a mean of 10.2 KB and a median of 5.5 KB. This size distribution indicates that most games in our dataset are compact, self-contained implementations suitable for educational and prototyping purposes, while still including complex examples exceeding 50 KB. The predominance of smaller files (under 20 KB) aligns with typical Pygame project patterns and ensures computational efficiency during evaluation while maintaining functional completeness.

Pygame Module Usage Patterns Figure 29 quantifies the frequency of core Pygame API usage across our dataset. The analysis shows that fundamental modules like `pygame.display` (91.5%) and `pygame.event` (68.3%) are nearly universal, confirming adherence to standard Pygame development patterns. The moderate usage of advanced features like `pygame.sprite` (21.3%) and `pygame.mixer` (19.2%) indicates a balanced representation of both basic and sophisticated game development techniques within our corpus.

Game Type Distribution The game type analysis in Figure 30 demonstrates substantial diversity in our dataset, with arcade games comprising the largest category (47.3%) followed by shooter games (17.7%)

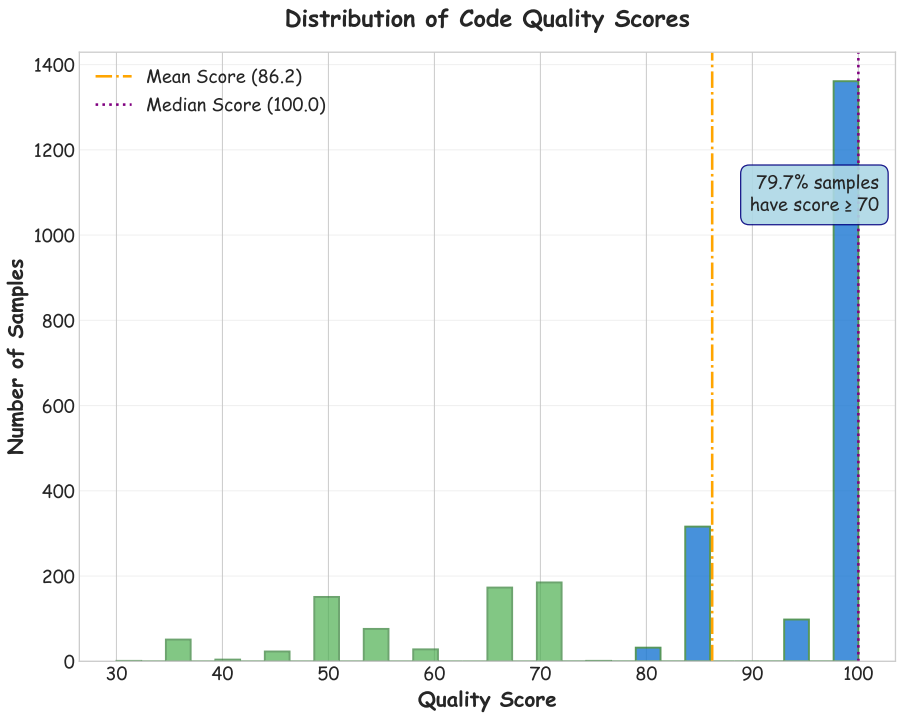


Figure 27: Distribution of code quality scores across 2,500 selected samples. And 2500 is the seed set selected after clustering, and 2219 is the final test set after LLM pipeline and manual verification.

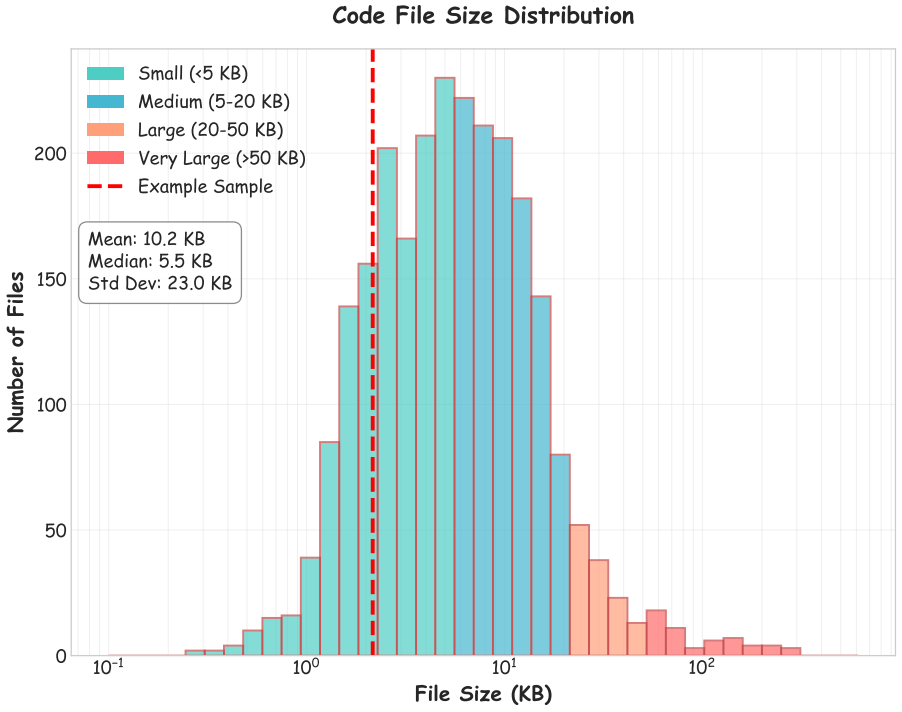


Figure 28: File size distribution showing log-normal characteristics with mean 10.2 KB and median 5.5 KB, indicating predominantly compact but complete implementations.

and other miscellaneous types (23.7%). This distribution reflects the natural prevalence of different game genres in the Pygame community while ensuring adequate representation of specialized categories like physics simulations, RPGs, and educational games. The balanced representation across game types enhances the benchmark's ability to evaluate diverse programming patterns and game mechanics.

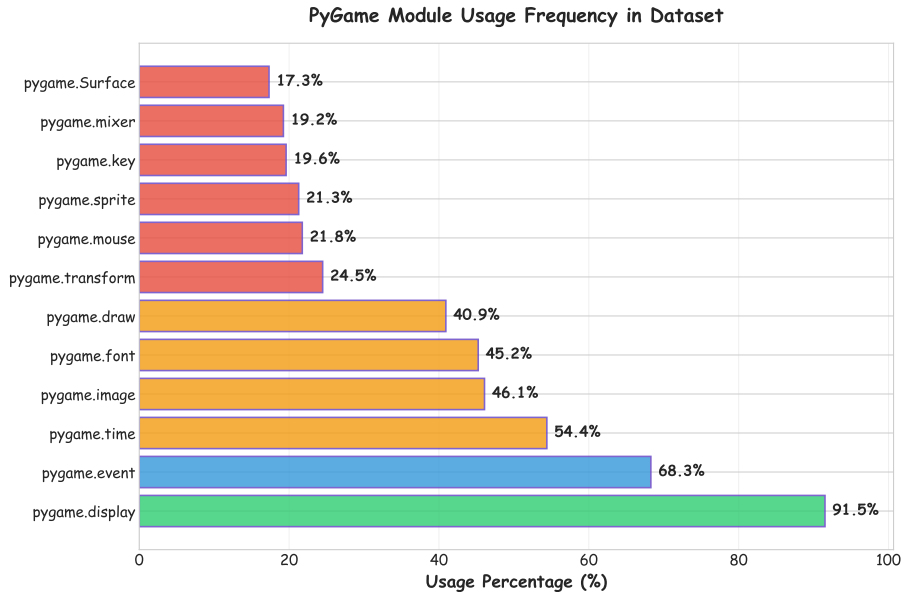


Figure 29: Frequency analysis of Pygame module usage, with core modules like display (91.5%) and event handling (68.3%) showing high adoption rates.

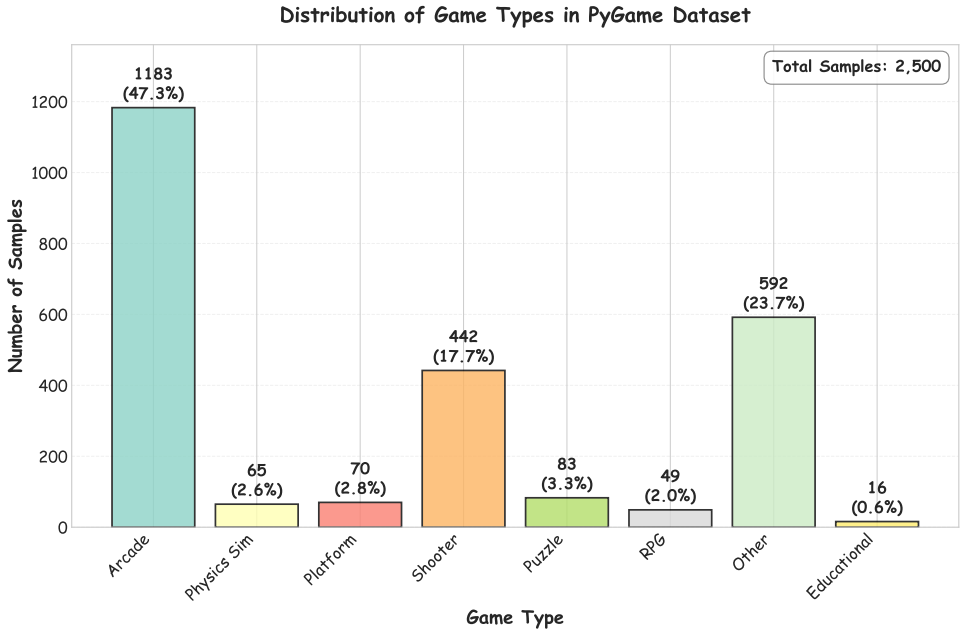


Figure 30: Distribution of game types in the dataset, representation across eight distinct genres.

Code Structure Analysis Figure 31 provides quantitative insights into the structural characteristics of our dataset. The average code file contains 12.0 functions, 2.2 classes, and 283.3 lines of code, indicating well-structured implementations that follow object-oriented programming principles. The prevalence of for loops (10.6 per file) and event handlers reflects the iterative and interactive nature of game programming, while the consistent presence of game loops and display updates confirms adherence to standard Pygame architectural patterns.

Cluster Quality Correlation The scatter plot in Figure 32 examines the relationship between cluster size and average quality scores. The weak positive correlation (0.138) suggests that larger clusters do not necessarily contain higher-quality code, validating our quality-based selection approach within each cluster. This analysis confirms that our methodology successfully identifies the best exemplars from each functional group regardless of the cluster's overall size, ensuring consistent quality across diverse game categories.

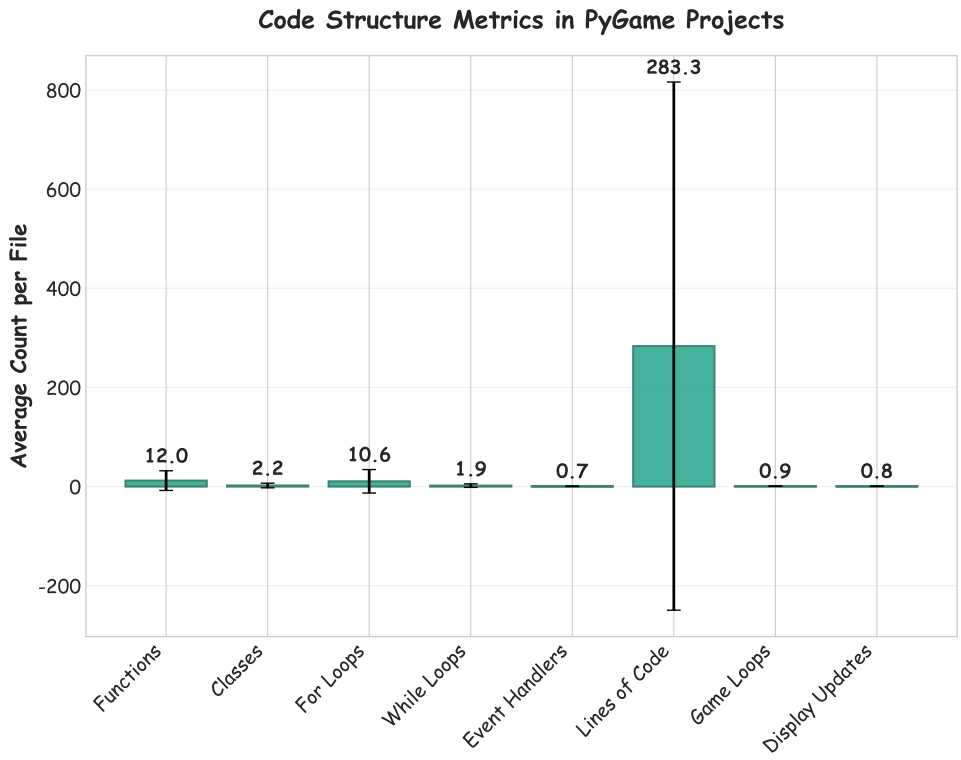


Figure 31: Structural metrics of code samples, showing average counts of functions (12.0), classes (2.2), and other programming constructs per file.

Figure 32: Correlation analysis between cluster size and average quality scores, showing weak correlation (0.138) that validates quality-based selection within clusters.

Complexity by Game Type Figure 33 reveals significant variation in code complexity across different game genres. Physics simulation and RPG games exhibit the highest complexity scores, reflecting their sophisticated mechanics and state management requirements. In contrast, educational and puzzle games show lower complexity, aligning with their focus on simplicity and clarity. This complexity distribution

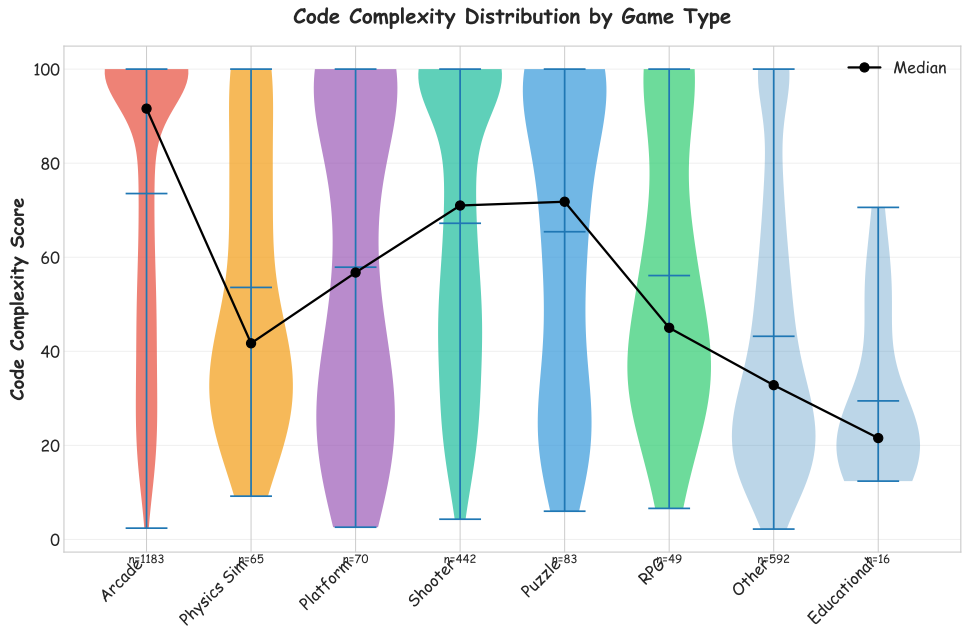


Figure 33: Box plot analysis of code complexity scores across game types, with physics simulations and RPGs showing highest complexity variance.

Figure 34: Radar chart analysis of quality components across different score tiers, highlighting strengths in structure and organization for high-quality samples.

ensures that our benchmark captures the full spectrum of programming challenges inherent in different game development domains.

Quality Components Radar Analysis The radar chart in Figure 34 provides a multi-dimensional view of quality factors across different performance tiers. High-quality samples (85+) consistently excel across all dimensions, particularly in basic structure, display setup, and code organization. The analysis reveals that documentation and error handling are key differentiators between quality tiers, while basic functionality components like pygame initialization and game loops are well implemented across all levels.

This comprehensive quality assessment ensures that our dataset maintains high standards while capturing diverse implementation approaches.

O System Architecture and Performance

Complete Pipeline Performance

End-to-End Workflow (Generation → Recording → Evaluation)

Pipeline Components:
1. **Code Generation**: OpenAI API with parallel processing
2. **Game Recording**: Optimized pygame execution with media capture
3. **Multi-modal Evaluation**: Code + Screenshot + Video analysis

Performance Optimizations:
- Async I/O for file operations
- Batch processing for efficiency
- Configurable worker pools
- Resume capability for interrupted runs
- Streaming API responses

Quality Assurance:
- Automatic retry mechanisms
- JSON validation with error handling
- Progress tracking and recovery
- Comprehensive logging and statistics

Overall Success

>80% end-to-end success rate from requirement to final evaluation score.

Scalability

Handles 1000+ games with configurable parallelization and resource management.

Reliability

Robust error handling with automatic recovery and detailed failure analysis.

P Game Code Generation Pipeline

Game Code Generation Case

User Request (Game Requirement 🔩 + System Prompt ✎ + Code Template ⇨)

🔩 Generate a complete pygame code based on the following game requirement:
"Create a simple Snake game where the player controls a snake to eat food and grow longer. The game should have collision detection and score display."

✎ System Prompt: "You are a pygame game development expert, good at quickly developing small games based on requirements."

Requirements:
1. Generate a complete and runnable pygame code
2. The game should automatically run for 10 seconds and then exit
3. Include all necessary import statements, especially 'import time'
4. Add time-based automatic exit mechanism
5. Add a visual timer showing elapsed time
6. Set reasonable FPS

⇨ Code Template Structure:

```python
import pygame
import time
start_time = time.time()
# In main loop:
current_time = time.time()
if current_time - start_time >= 10:
    running = False
```

Generated Code

Complete pygame Snake game with automatic exit, timer display, and proper game loop implementation.

Success Rate

85.3% of generated codes compile and run successfully.

Avg Generation Time

2.3 seconds per game with parallel processing.

Q Game Recording and Media Capture

Game Recording Pipeline

Recording Process (Code Execution 🔩 + Screenshot Capture ✎ + Video Recording 🔊)

🔩 Execute generated pygame code with optimized performance:
- Record duration: 10-30 seconds
- Video FPS: 3-30 (configurable)
- Screenshot format: JPG (faster) or PNG
- Async I/O for better performance

✎ Screenshot Capture at specific timestamps:

```python
_screenshot_times = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# Capture at each second for consistent evaluation
```

Optimized surface conversion:

```python
def _pygame_surface_to_cv2_optimized(surface):
    raw = pygame.image.tobytes(surface, 'RGB')
    img = np.frombuffer(raw, dtype=np.uint8)
    return cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
```

🔊 Video Generation with optimized settings:
- Use mp4v codec for fast encoding
- Batch write frames for efficiency
- Configurable frame interval based on target FPS
- Background thread for async I/O operations

Output Media

10 screenshots + 1 gameplay video per game with consistent quality.

Processing Speed

Average 1.2s per game with 20 parallel workers.

Success Rate

92.7% games successfully recorded with complete media files.

R Multi-Modal Game Evaluation System

Game Evaluation Framework

Evaluation Components (Code Analysis 🔩 + Screenshot Review ✎ + Video Assessment 🔊)

🔩 Code Quality Evaluation (0-100 points):
- Functionality (0-25): Implementation completeness
- Code Quality (0-25): Structure and readability
- Game Logic (0-25): Logic correctness
- Technical Implementation (0-25): pygame usage efficiency

✎ Visual Quality Assessment from Screenshots (0-100 points):
- Visual Completeness (0-25): UI elements presence
- UI Design (0-25): Layout and visual effects
- Function Display (0-25): Key features visibility
- Overall Quality (0-25): Visual completion level

Up to 20 screenshots analyzed per game for comprehensive coverage.

🔊 Dynamic Behavior Analysis from Video (0-100 points):
- Animation Effect (0-25): Smoothness and naturalness
- Interaction Logic (0-25): User input responsiveness
- Game Flow (0-25): Gameplay continuity
- Dynamic Quality (0-25): Overall playability

Final Score

Average of three evaluation components with automatic retry mechanism for robust scoring.

Retry Mechanism

Up to 10 attempts with JSON parsing validation for reliable results.

Evaluation Speed

4 parallel processes with streaming responses for efficient processing.