

ByteSized32Refactored: Towards an Extensible Interactive Text Games Corpus for LLM World Modeling and Evaluation

Haonan Wang¹, Junfeng Sun², Xingdi Yuan³, Ruoyao Wang⁴, Ziang Xiao¹

¹Johns Hopkins University ²Liaoning Technical University ³Microsoft Research Montréal

⁴Central University of Finance and Economics

hwang298@jh.edu junf1831@outlook.com eryua@microsoft.com

wangruoyao@cufe.edu.cn ziang.xiao@jhu.edu

Abstract

Simulating interactive world models remains a core challenge in Large Language Models (LLMs). In this work, we introduce the **ByteSized32Refactored**¹, a refactored, modular, and extensible implementation of the original ByteSized32² corpus to explore the task of text game generation. We further optimize the code structure of each text game and create the `GameBasic.py` foundation library, which centralizes common logic across all 32 games by abstracting 7 base classes (`GameObject`, etc.) into reusable modules, thereby reducing from **20k** to **10k** total lines of Python code compared to the original Bytesized32. Our refactored implementation enables extendability - with our centralized design, **ByteSized32Refactored** can be more efficiently extended to include text games of new scenarios and specifications by reusing the shared logic and functionalities. Extensive experiments with GPT-4o demonstrate a mix of performance - with **Byte-sized32Refactored**, the generated text games for unseen scenarios showcase quality improvements on two of the four evaluation dimensions while decreases on the other two, indicating that the hierarchical structure of the refactored code presents new challenges for LLMs. Overall, we highlight that our extensible code structure, centered on the foundation library and the modular optimization, not only facilitates LLM adaptation to environment specifications but also establishes a scalable environment that supports future extensions.

1 Introduction

Human intelligence and problem-solving are characterized by the ability to understand and interact with structured representations of the world, which support reasoning, planning, and generalization

across complex environmental tasks (Gignac and Szodorai, 2024; Barsalou, 1999). Large language models (LLMs) have demonstrated impressive capabilities in natural language processing and zero-shot performance across task (Kojima et al., 2022). While these abilities indicate a nascent capacity for LLMs to possess the foundations of world modeling, their reasoning remains largely textual and lacking explicit formalization. Using LLMs to play text games primarily assesses their reasoning under constrained specifications, rather than whether they maintain consistent and structured models of the world (Jansen and Côté, 2022; Latitude Games, 2019; Hausknecht et al., 2019). To move beyond surface-level evaluation, we argue that generating interactive world simulators in text games provides a more suitable medium for examining world models in LLMs. It offers a structured, interpretable, and testable representation of environments as they formalize objects, states, constraints, and goals with natural language description (Wang et al., 2023, 2022). ByteSized32 (Wang et al., 2023, 2022) marked an advanced step in formalizing world modeling, which focuses on the challenge of building text-based environments, where players interact with the world through natural language descriptions and commands. Such text-based environments enable agents to effectively operate within interactive text games to accomplish goals in complex scenarios through natural language input. Its highlights make it an ideal testing environment for evaluating the ability of an AI system in understanding, manipulating and constructing structured representations of the world. However, despite its significance, ByteSized32 also presents notable limitations in extendable code structure design: each game is implemented as a standalone file with logic, which hampers extensibility (adding new actions/objects/rules) and better input formulation for LLMs, i.e. flexible single-shot and multi-shot evaluation within a fixed context window, to generate

¹ByteSized32Refactored: <https://github.com/isle-dev/BYTESIZED32-Refactored>

²ByteSized32: <https://github.com/cognitiveailab/BYTESIZED32>

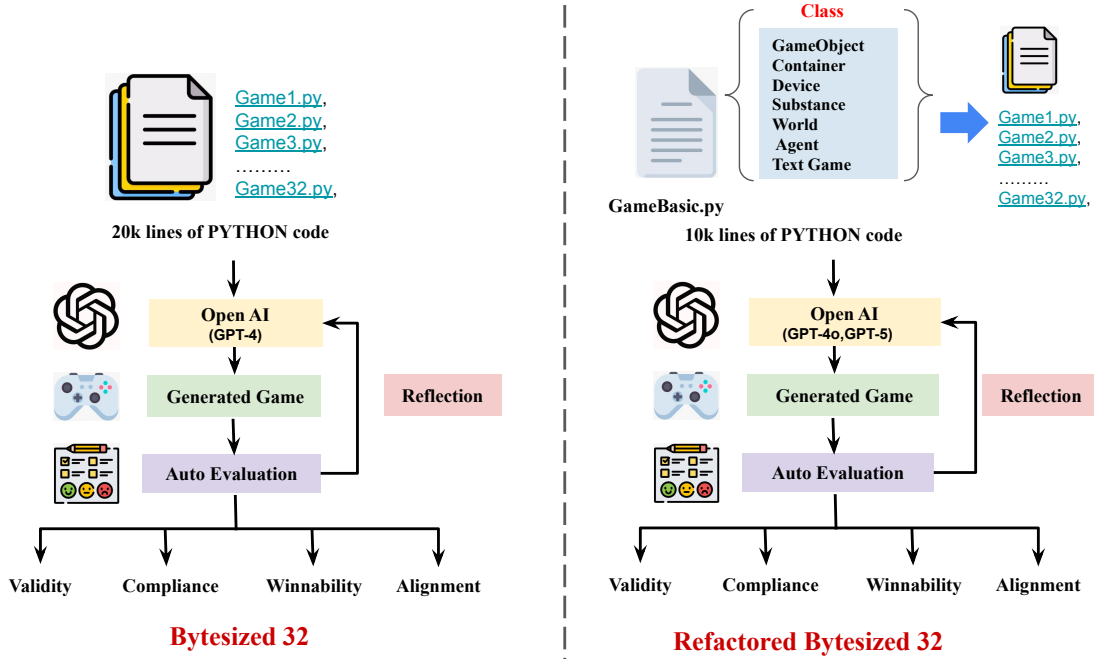


Figure 1: An overview of text game generation and evaluation process between the **ByteSized32Refactored** and ByteSized32. GPT-4o and GPT-5 generate games using in-context learning with a prompt consisting of (1) a single template example game, and (2) the task specification for the target game to generate. Generated games are then self-reflected by providing the models with error outputs from a PYTHON interpreter that detects syntactic and API issues. Each generated game is subsequently evaluated by an automated evaluation for technical validity, specification compliance, physical reality alignment, and winnability.

new text games. In this work, we present **ByteSized32Refactored**, a restructured, extensible text game corpus built upon the original **ByteSized32**, designed to evaluate the ability of multiple large language models (LLMs) like GPT-4o³ to construct and test task-specific world environments. Specifically, compared with the original **ByteSized32**, by reorganizing the foundation library `GameBasic.py` that abstracts the 7 base classes (e.g. *GameObject*, *Container*, *World*, *Agent*, and *TextGame*, etc.) and optimizing the code details of each game file. As a result, each game in **RefactoredByteSized32** now only implements domain-specific objects and task logic, while all shared functionality is reused from `GameBasic.py`, ensuring modularity, extensibility, and reproducibility across the corpus. Moreover, the refactorization enhances the extendability of **ByteSized32** to incorporate new text games with novel specifications, improves the utility of existing text games as demonstrations for in-context game generation, and presents a new challenge to LLMs for code generation with hierarchical structures.

The contributions of this work are as follows:

1. We present **ByteSized32Refactored**, a modu-

lar refactorization of the original **ByteSized32** text game corpus (expressed as common-sense text games in Python). In contrast to the original **20k** lines monolithic implementation, our refactored corpus further optimized each game by replacing verbose control flows with dictionary-based dispatch, streamlining string operations, and abstracting redundant logic to comprise **10k** lines (subsection 3.2.2).

2. We develop `GameBasic.py` (subsection 3.2.1, Figure 2), a foundational library that provides a shared abstraction layer for all 32 text games, ensuring structural consistency, reducing redundant code, and enabling scalable extensions for future development.
3. We evaluate GPT-4o on the refactored and original corpus, providing a fair basis for measuring progress over prior results (section 4).

2 Related Work

2.1 Text Game

Text games, also known as interactive fiction (IF) environments, simulate interactive worlds in which both observations and actions are expressed entirely in natural language. These text games

³<https://openai.com/index/gpt-4o-system-card/>

costs much less than for 2D or 3D games and provides natural language game descriptions of the current state(Jansen, 2021; Li et al., 2021; Nelson, 2006; Wang et al., 2025), which require users or agents to understand and generate natural language commands to interact with the game environment, which have been widely used as a challenging testbed for natural language processing to evaluate multiple capabilities of AI systems(Côté et al., 2018; Wang et al., 2023, 2022; Jansen and Côté, 2022; Wang et al., 2024; Shridhar et al., 2020). Two dominant approaches have emerged in evaluating AI systems to navigate and complete text world games: reinforcement learning **(RL)-based agents** modeling the environment as a POMDP (Kaelbling et al., 1998) and large language model **(LLM)-based agents**, where observations are input to the LLMs and the outputs are executed as actions(Cui et al., 2025). For **(RL)-based agents**, prior works have explored various techniques to enhance the learning process and performance for non-LLM-based agents. Subsequent efforts augmented RL agents with structured knowledge representations, such as knowledge graphs, to improve state tracking and filter irrelevant actions(Narasimhan et al., 2015; Hausknecht et al., 2020; Ammanabrolu and Riedl, 2018; Yuan et al., 2018; Murugesan et al., 2021; Ryu et al., 2023). More recent studies have introduced advanced mechanisms, including graph attention layers for more efficient action selection(Ammannabrolu and Hausknecht, 2020), role-playing agents that intrinsically reward adherence to personas(Peng et al., 2023), and soft prompts to enable a single agent to adopt multiple personas(Cui et al., 2023) flexibly. In addition to these purely RL-based methods, recent work has explored hybrid approaches that integrate LLMs with RL agents.(Basavatia et al., 2024)employ LLMs to procedurally generate novel text game environments, thereby enabling RL agents to be tested on generalization across unseen tasks. Similarly, (Golchha et al., 2024) utilizes LLMs to provide decision-level guidance to RL agents, improving their reasoning and action selection.

2.2 LLMs for Code Generation

Large Language models (LMs) are rapidly being deployed in commercial applications, and several recent works for evaluated the ability of LLMs to generate executable program code(Dong et al., 2025; Huynh and Lin, 2025). The current standard benchmarks HumanEval(Chen et al., 2021),

MBPP(Austin et al., 2021), and APPS(Hendrycks et al., 2021) are in a longstanding pursuit of synthesizing code from natural language descriptions(Yu et al., 2018; Li et al., 2022; Zan et al., 2022). Similarly, SWE-bench(Jimenez et al., 2024) has advanced the field by framing software engineering tasks in more realistic contexts, requiring models to navigate large codebases, generate patches, and capture dependency-based relationships across modules. LiveCodeBench(Jain et al., 2024) evaluates models on real-world code changes from open-source projects, while RepoBench(Liu et al., 2023b) focuses on repository-level code completion across multiple files. BigCodeBench(Zhuo et al., 2024) emphasizes compositional reasoning by requiring function calls across a wide range of libraries, and CoCo-Bench(Yin et al., 2025)extends evaluation to tasks such as code understanding and review. In addition to these benchmarks, several works have explored a variety of extensions have been proposed to broaden its coverage. These include multi-language variants(Cassano et al., 2022; Athiwaratkun et al., 2022; Orlanski et al., 2023), modifications to the edit scope and task granularity(Yu et al., 2024; Du et al., 2023), as well as novel code completion benchmarks(Muennighoff et al., 2023; Liu et al., 2023a). Other efforts introduce alternative coding paradigms(Yin et al., 2022; Yang et al., 2023). However, most of these benchmarks remain limited to short code snippets and fail to capture the complexity of executable programs. They rarely treat code as a fully interactive environment for simulating the world.

3 Bytesized32Refactored Corpus

3.1 Corpus Overview

Similar to the original BYTESIZED32, the **RefactoredByteSized32** also contains 32 common-sense task-specific text games like *washing dishes*, *boiling water*. Each text game is accompanied by a **task specification**, which provides a structured summary of the interactive environment and game objectives, including:

- **Task Description:** The natural language description of the task goal the agent must achieve, (e.g., *washing dirty dishes using a dishwasher*).
- **Task-Critical Objects:** The set of objects indispensable for solving the task (e.g., *dishes*, *dish soap*, *dishwasher*).
- **Actions:** The valid operations available to the agent, such as *opening/closing* containers.

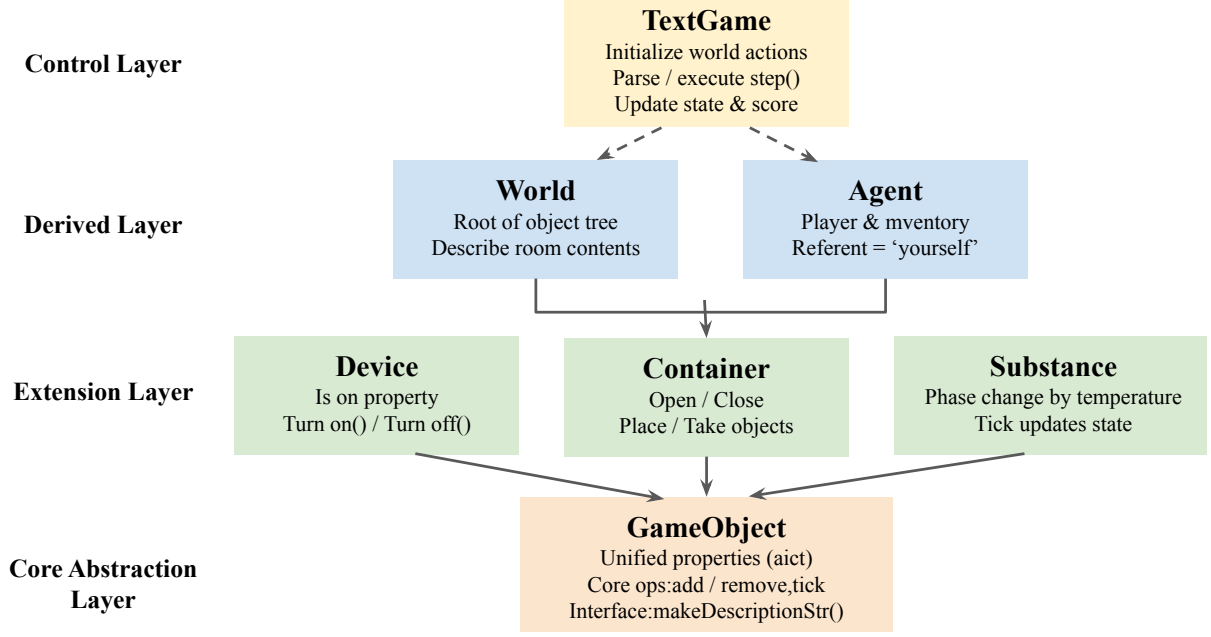


Figure 2: Layered architecture of the GameBasic library. Lower layers provide unified abstractions and reusable modules, while the Derived Layer defines the world and the agent. At the top, TextGame orchestrates these entities to execute the game. Arrows denote inheritance or dependency, and specifically a control relation from TextGame to World and Agent.

- **Distractors:** Objects or actions that are irrelevant to the task or deliberately increase its difficulty (adding *food* that creates additional dirty dishes).
- **Solution:** A high-level procedural outline describing a canonical way to complete the task, such as opening the dishwasher.

3.2 Refactored Code Structure

Each task specification of the text game encodes a goal-oriented task that requires agents to manipulate objects, containers, and devices through naturalistic action sequences shown in Figure 5 and Table 1. It is worth noting that we show in Table 2 the number of examples that can fit in the context window of various LLMs. Compared with Byte-sized32, the **Bytesized32Refactored** corpus allows substantially more examples to be packed into the prompt as demonstrations under different context limits.

3.2.1 GameBasic.py Library

In the original ByteSized32 implementation, each game file embedded the complete implementation of multiple base classes such as *GameObject*, *Container*, *World*, *Agent*, and *TextGame*. While this approach ensures code independence, it results in significant maintenance costs. We developed GameBasic.py in **RefacoredByteSized32**, a centralized abstraction library that encapsulates common components into a modular code structure

shown in Figure 2. Instead of these original classes in each game file, the GameBasic.py provides a unified interface and shared logic for core functionalities, which defines seven base class abstractions: *GameObject* (root class for all entities), *Container* and *Device* (interactive objects), *Substance* (physical state modeling), *World* and *Agent* (environment and player), and *TextGame* (template for task execution) by inheriting from these base classes. Compared to the code structure of the original ByteSized32, the GameBasic.py brings two core contributions to the RefacoredByteSized32:

- **Consistency:** All games inherit from a unified parent class interface, ensuring structural uniformity. Developers can implement task-specific logic by simply extending these base classes, significantly reducing development complexity. These shared components in GameBasic.py are centralized and maintained in a single location, drastically reducing repetitive code and improving maintainability.
- **Flexibility and Scalability:** Not only provides generic class implementations but also reserves function interfaces (e.g., `initializeWorld()`, `getTaskDescription()`). This design enables each game to implement customized logic with minimal development overhead, thereby enhancing extensibility and adaptability for future tasks.

Table 1: Comparison of BYTESIZED32 and Refactored Game Statistics (per game)

Statistic	BYTESIZED32	BYTESIZED32Refactored	Δ
Lines of Python code	618.1	303.19	-50.9
Lines of comments	198.1	198.1	0
Tokens per Game	6792	2896	-3896
Action verbs	9.8	9.8	0
Valid actions	306.6	306.6	0
Object classes	5.4	5.6	+0.2
Object instances	7.4	6.5	-0.9
Expert path length	12.8	12.8	0

3.2.2 Code Modularity Optimization

In addition to the development of `GameBasic.py`, our refactoring emphasized structured logic and loop mechanisms, leveraging higher-level function encapsulation and abstraction to replace repetitive implementations, underwent further code optimization like object initialization and action handling:

- **Action Parsing with `action_map`:** In the original `ByteSized32`, the `step()` function relied on lengthy `if/elif` chains to dispatch actions. After the refactored code replaces this with an `action_map` dictionary, significantly improving extensibility and simplicity by mapping actions directly to their corresponding handlers.
- **Efficient String Construction:** In the original `ByteSized32`, `makeDescriptionStr()` relied on verbose string concatenation, whereas the refactored version uses `join()` and conditional expressions for cleaner and more efficient code.

4 Experiment

4.1 Experiment Setup

We demonstrate the utility of the **ByteSized32Refactored** compared with the original `ByteSized32` by evaluating the quality of games generated by SOTA large language model (i.e., GPT-4o) on both corpora. Specifically, each evaluation task provides the model with 1) a one-shot example of a game in the corpus; 2) a task specification drawn from an unseen evaluation set that asks the model to generate a new game following the specification.

We follow the original `ByteSized32` to randomly select the one-shot example and adopt the same prompt for task specification. However, while each example is presented as a single code snippet in the original corpus, in **ByteSized32Refactored**, we present the one-shot example with 1) the code and corresponding descriptions of `GameBasic.py`

that informs the model of the unified interface and shared logic for game construction, and 2) the actual game code of the example showcasing how to properly leverage the classes in `GameBasic.py` to generate a game. We provide complete details of prompts and models in Figure F and Appendix A, respectively.

4.2 Evaluation Metrics

To ensure fair comparison with the original `ByteSized32`, we follow the original `ByteSized32` to assess the quality of generated games from the following four dimensions:

Technical Validity assesses whether core mechanisms of the game run error in Python, which includes environment initialization, valid action generation, and state updates. We explore a trajectory-based method that action sequences from the initial state up to depth three (max 100 actions each) to detect runtime errors and logic inconsistencies, thereby ensuring stability and systematic verification.

Specification Compliance measures whether a generated game satisfies its task requirements by verifying the presence and correctness of required objects, actions, and other elements. Compliance is evaluated through automatic matching against the task specification, ensuring fidelity to the intended design.

Physical Reality Alignment evaluates whether game actions respect basic physical constraints (e.g., opening a container before placing objects inside). Using trajectory-based exploration up to depth three with 100 sampled trajectories, we evaluate each step through binary judgments with justifications, ensuring logical grounding in real-world principles.

Winnability focuses on whether the generated game is winnable, i.e., whether at least one complete sequence of actions can lead to successful

Table 2: Number of examples that can fit in the context window of various LLMs. Compared with Bytesized32, Bytesized32Refactored can fit more examples into the prompt as demonstrations under various context limits.

	GPT-4o	Qwen3-32B	Llama3.2-1b-Q
Context Length	128k	32k	8k
Bytesized32	18	4	1
Bytesized32Refactored	44	9	2

task completion in the game.

5 Results and Analysis

We evaluate all generated games by GPT-4o on **ByteSized32Refactored** and the original Bytesized32, reporting results before and after self-reflection to investigate whether our providing refactorization meaningfully change how LLMs build world modeling.

Table 3: Technical validity of GPT-4o on **ByteSized32Refactored** across reflection steps (0–3)

Technical Validity Measurement	Number of Reflections			
	0	1	2	3
Game Initialization	56.25%	81.25%	85.42%	85.42%
Vaild Actions	55.21%	62.50%	69.79%	70.83%
Runnable Game	17.71%	39.58%	55.21%	61.46%

Table 4: Technical validity of GPT-4o on **Bytesized32** across reflection steps (0–3)

Technical Validity Measurement	Number of Reflections			
	0	1	2	3
Game Initialization	91.67%	94.79%	95.83%	95.83%
Vaild Actions	75.00%	90.62%	90.62%	90.62%
Runnable Game	48.96%	75.00%	80.21%	82.29%

Technical Validity The results of the technical validity are summarized in Table 3 and Table 4. On all of *game initialization*, *valid actions* and *runnable game*, the generated games demonstrate increasing quality with more founds of reflection on both **Bytesized32Refactored** and the original Bytesized32, showcasing the effectiveness of reflection in improving the quality of game generation across original and refactored codebase. However, on all of the three metrics, the game quality in **Bytesized32Refactored** down-performs the counter-part in the original Bytesized32 across all

rounds of reflection. The degradation of quality reveals that the refactored code presents new challenges for LLM to reasoning over the hierarchical code structure and perform more complex class-based code generation.

Specification Compliance and Winnability We report the specification compliance and winnability performance of GPT-4o under the refactored and original corpus in Table 5 and Table 6. Specif-

Table 5: Specification compliance and winnability for GPT-4o on **ByteSized32Refactored** before/after reflection.

Measurement	Reflection		Δ
	Before	After	
<i>Specification Compliance</i>			
Task-critical objects	100.0%	100.0%	0.0%
Task-critical actions	93.75%	93.75%	0.0%
Distractors	31.25%	28.12%	-3.13%
Winnability	33.3%	54%	+20.7%

Table 6: Specification compliance and winnability for GPT-4o on **ByteSized32** before/after reflection.

Measurement	Reflection		Δ
	Before	After	
<i>Specification Compliance</i>			
Task-critical objects	100.0%	100.0%	0.0%
Task-critical actions	90.62%	90.62%	0.0%
Distractors	56.25%	53.12%	-3.13%
Winnability	20%	34%	+14%

ically, on both corpus, the compliance of task-critical objects and actions and winnability increase with three rounds of reflection whereas the number of distractors slightly drops. Across initial generation and reflections, the generated game on **Bytesized32Refactored** consistently outperforms games on the original counterpart over object/action compliance and winnability, showcasing that GPT-4o is able to generate more reasonable games that better comply with specifications on **Byte-**

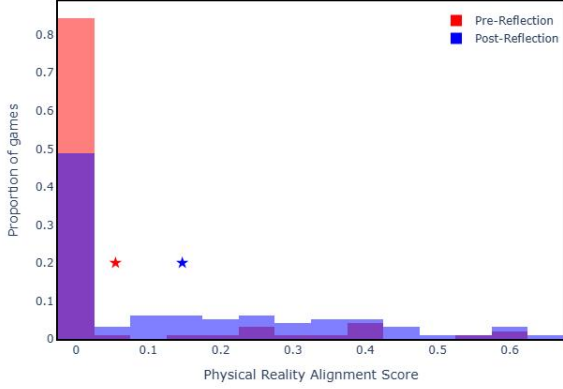


Figure 3: Histogram of physical reality alignment scores for GPT-4o on **ByteSized32Refactored** before (red) and after reflection (blue). Asterisk represent average scores (0.055 pre-reflection, 0.147 post-reflection)

sized32Refactored, highlighting the effectiveness of our refactorization.

Physical Reality Alignment Figure 3 and Figure 4 illustrate how the generated games align with physical reality on both corpora. On average, comparing with performance on the original corpus, GPT-4o struggles to output physically-aligned games on **Bytesized32Refactored** on its initial generation but can attain a similar level of alignment after reflection. By comparing the two histograms, we observe that the higher abstraction and compressed structure of Refactored ByteSized32 initially make it more challenging for GPT-4o to capture physical consistency. This is evident in the distribution, which initially clusters near zero with few mid- or high-scoring samples. However, reflection significantly reduces low scores and gradually shifts the distribution toward the mid- and high-range. The results indicate that with reflection, the refactored corpus does not hinder GPT-4o from yielding physically-aligned text games compared to the original corpus.

Summary of results In summary, our findings across the four evaluation dimensions highlight two key insights in the evaluation of LLMs for text-game generation: (1) The abstracted interfaces in `GameBasic.py` pose a greater challenge for GPT-4o than the redundant implementation in the original corpus, which embeds more explicit action cues that enables models to pass validity checks without deep reasoning. By contrast, abstraction increases reasoning demands and raises the generation barrier, making GPT-4o more prone to omitting nec-

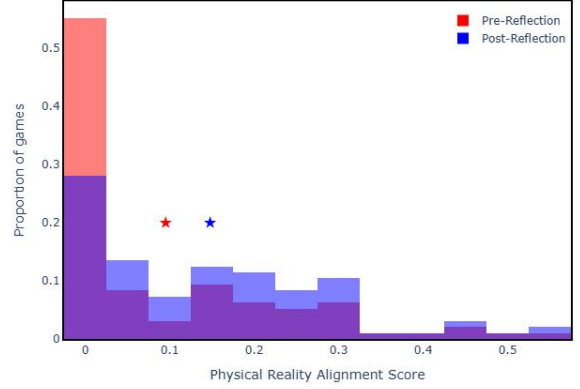


Figure 4: Histogram of physical reality alignment scores for GPT-4o on **ByteSized32** before (red) and after reflection (blue). Asterisk represent average scores (0.095 pre-reflection, 0.148 post-reflection)

essary components or producing invalid structures during initial generation. (2) On the other hand, our hierarchical abstractions enables LLMs to generate valid text games that 1) are much more complied with given specification; 2) possess high-quality underlying game logic to ensure winnability; and 3) do not sacrifice the alignment with physical reality under reflection.

6 Conclusion

In this work, we presented **ByteSized32Refactored**, a modular and extensible reimplement of the original **ByteSized32** corpus, designed to advance the exploration of text game generation with large language models (LLMs). `GameBasic.py` not only reduces code redundancy in text-based games, but also decouples basic components, such as `gameobject`, `container`, and `agent`, from specific task logic, providing a unified framework for future text-based games. This design makes common text-based game features easy to extend and opens up the possibility of unlimited future expansion. With our refactored design, **ByteSized32Refactored** can be easily extended to new game scenarios by reusing common game logic and substantially increasing the number of examples in a limited LLM context window. Our experiments with GPT-4o demonstrate mixed performance of the generated games, revealing both the effectiveness of the refactored codebase and new challenges presented by the hierarchical design. Overall, our findings emphasize both the promise and limitations of modular design and foundational

libraries in enabling LLMs to handle complex environment specifications, paving the way for future progress at the intersection of world modeling and code generation.

Limitations

Our work has several limitations that should be acknowledged.

1. The **ByteSized32Refactored** corpus alters token distribution and code structure. This raises the possibility that observed improvements stem from structural biases rather than genuine advancements in modeling ability.
2. Our evaluation metrics—validity, compliance, alignment, and winnability—serve as approximations of world modeling but have limited construct validity, being susceptible to framework biases and coverage gaps.
3. While reflection enhances measurability, it does not directly improve ability. Future work should aim to strengthen validity through invariant testing and randomization, disentangle compliance dimensions, clarify alignment with multi-judge evaluations, and expand winnability metrics with broader coverage and fuzzing techniques.
4. Finally, we perform reflection by regenerating the complete program at each step and target only a single error at a time. This process could be made more efficient by outputting only a code diff and batching multiple errors at once.

References

- Prithviraj Ammanabrolu and Matthew Hausknecht. 2020. Graph constrained reinforcement learning for natural language action spaces. *arXiv preprint arXiv:2001.08837*.
- Prithviraj Ammanabrolu and Mark O Riedl. 2018. Playing text-adventure games with graph-based deep reinforcement learning. *arXiv preprint arXiv:1812.01628*.
- Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, and 1 others. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.
- Lawrence W Barsalou. 1999. Perceptual symbol systems. *Behavioral and brain sciences*, 22(4):577–660.
- Shreyas Basavatia, Keerthiram Murugesan, and Shivam Ratnakar. 2024. Starling: Self-supervised training of text-based reinforcement learning agent with large language models. *arXiv preprint arXiv:2406.05872*.
- 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. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Marc-Alexandre Côté, Akos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, and 1 others. 2018. Textworld: A learning environment for text-based games. In *Workshop on Computer Games*, pages 41–75. Springer.
- Christopher Cui, Xiangyu Peng, and Mark Riedl. 2023. Thespian: Multi-character text role-playing game agents. *arXiv preprint arXiv:2308.01872*.
- Christopher Zhang Cui, Xingdi Yuan, Ziang Xiao, Prithviraj Ammanabrolu, and Marc-Alexandre Côté. 2025. Tales: Text adventure learning environment suite. *arXiv preprint arXiv:2504.14128*.
- Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. [A survey on code generation with llm-based agents](#). *Preprint*, arXiv:2508.00083.
- Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.
- Gilles E Gignac and Eva T Szodorai. 2024. Defining intelligence: Bridging the gap between human and artificial perspectives. *Intelligence*, 104:101832.
- Hitesh Golchha, Sahil Yerawar, Dhruvesh Patel, Soham Dan, and Keerthiram Murugesan. 2024. Language guided exploration for rl agents in text environments. *arXiv preprint arXiv:2403.03141*.
- Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7903–7910.
- Matthew J. Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2019. [Interactive fiction games: A colossal adventure](#). In *AAAI Conference on Artificial Intelligence*.

- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.
- Nam Huynh and Beiyu Lin. 2025. [Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications](#). *Preprint*, arXiv:2503.01245.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Live-codebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.
- Peter A Jansen. 2021. A systematic survey of text worlds as embodied natural language environments. *arXiv preprint arXiv:2107.04132*.
- Peter Alexander Jansen and Marc-Alexandre Côté. 2022. [Textworldexpress: Simulating text games at one million steps per second](#). *ArXiv*, abs/2208.01174.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. [Swe-bench: Can language models resolve real-world github issues?](#) *Preprint*, arXiv:2310.06770.
- Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.
- Latitude Games. 2019. Ai dungeon. <https://github.com/latitudegames/AIDungeon>. Accessed: 2025-08-30.
- Belinda Z Li, Maxwell Nye, and Jacob Andreas. 2021. Implicit representations of meaning in neural language models. *arXiv preprint arXiv:2106.00737*.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. [Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation](#). *Preprint*, arXiv:2305.01210.
- Tianyang Liu, Canwen Xu, and Julian McAuley. 2023b. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.
- Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 workshop on instruction tuning and instruction following*.
- Keerthiram Murugesan, Mattia Atzeni, Pavan Kapani-pathi, Kartik Talamadupula, Mrinmaya Sachan, and Murray Campbell. 2021. Efficient text-based reinforcement learning by jointly leveraging state and commonsense graph representations. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*, volume 2, pages 719–725. Association for Computational Linguistics.
- Karthik Narasimhan, Tejas Kulkarni, and Regina Barzilay. 2015. Language understanding for text-based games using deep reinforcement learning. *arXiv preprint arXiv:1506.08941*.
- Graham Nelson. 2006. Natural language, semantic analysis, and interactive fiction. *IF Theory Reader*, 141(99):104.
- Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*, pages 26619–26645. PMLR.
- Xiangyu Peng, Christopher Cui, Wei Zhou, Renee Jia, and Mark Riedl. 2023. Story shaping: Teaching agents human-like behavior with stories. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pages 326–336.
- Dongwon Kelvin Ryu, Meng Fang, Shirui Pan, Gholamreza Haffari, and Ehsan Shareghi. 2023. A minimal approach for natural language action space in text-based games. *arXiv preprint arXiv:2305.04082*.
- Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.
- Haonan Wang, Mingjia Zhao, Junfeng Sun, and Wei Liu. 2025. Design and optimization of reinforcement learning-based agents in text-based games. *arXiv preprint arXiv:2509.03479*.
- Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. 2022. [Scienceworld: Is your agent smarter than a 5th grader?](#) *Preprint*, arXiv:2203.07540.
- Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. 2024. Can language models serve as text-based world simulators? *arXiv preprint arXiv:2406.06485*.

Ruoyao Wang, Graham Todd, Eric Yuan, Ziang Xiao, Marc-Alexandre Côté, and Peter Jansen. 2023. [Byte-sized32: A corpus and challenge task for generating task-specific world models expressed as text games](#). *Preprint*, arXiv:2305.14879.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. 2023. [Intercode: Standardizing and benchmarking interactive coding with execution feedback](#). *Preprint*, arXiv:2306.14898.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. 2022. [Natural language to code generation in interactive data science notebooks](#). *Preprint*, arXiv:2212.09248.

Wenjing Yin, Tianze Sun, Yijiong Yu, Jiawei Fang, Guangyao Su, Jiancheng Wang, Zekun Wang, Wei Wang, Ran Chen, Ziyun Dai, and 1 others. 2025. Coco-bench: A comprehensive code benchmark for multi-task large language model evaluation. *arXiv preprint arXiv:2504.20673*.

Hao Yu, Bo Shen, Dezhi Ran, Jiabin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Xingdi Yuan, Marc-Alexandre Côté, Alessandro Sordani, Romain Laroche, Remi Tachet des Combes, Matthew Hausknecht, and Adam Trischler. 2018. Counting to explore and generalize in text-based games. *arXiv preprint arXiv:1806.11525*.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*.

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 Model Setting

In this work, we make extensive use of OpenAI’s API. In all our experiments, we keep the following hyperparameters constant: In this work, we make

extensive use of OpenAI’s API. In all our experiments, we keep the following hyperparameters constant:

A.1 GPT-4o Model setting

- temperature=0.0
- top-p=1
- frequency-penalty=0.0
- persence-penalty=0.0

A.2 GPT-5 Model setting

- temperature=1
- top-p=1
- frequency-penalty=0.0
- persence-penalty=0.0

B An Example of Playthrough in Bytesized32Refactored Corpus

Playthrough of a training set game

Task Description: Your task is to feed the cat.

Observation: You find yourself in a kitchen. In the kitchen, you see:

- yourself
- a bowl
- a can of cat food
- a cat

Type 'help' for a list of possible actions.

→take bowl

The bowl is removed from the kitchen. You put the bowl in your inventory.

→take cat food

The cat food can is removed from the shelf. You put the cat food can in your inventory.

→open cat food

The cat food can is opened.

→put cat food in bowl

Cat food is placed in the bowl.

→put bowl on floor

The bowl is placed on the floor.

→feed cat

The cat eats the food.

Game completed.

Figure 5: An example of Playthrough.

C GPT-5 Experiment error and analysis

```

In [ ]: {
  "idx":0,
  "evaluation":"no",
  "short_justification":"The engine required holding the "
                        "accessible butter to put it into/onto "
                        "the pot, despite manipulation without "
                        "prior pickup being allowed, so the "
                        "responses aren't physically accurate."
}
{
  "idx":0,
  "evaluation":"yes",
  "short_justification":"Requiring holding the butter before "
                        "placing it in/on the pot is physically "
                        "sensible, and all observations"
}
{
  "idx": are0 consistent,
  "with theevaluation":"yes world",
  "short_justification":" stateIt." is
}

physically consistent to require holding the butter before
e placing it in or on the pot, and all observations align
with a realistic state.

"){"
  idx":0,"
  evaluation":"yes",
  short_justification":"
  The engine sensibly requires holding the butter before
  placing it and returns no errors or physically impossible
  outcomes.
"}{"
  idx":0,"
  evaluation":"yes",
  short_justification":"
  All responses are consistent with real-world constraints
  (you must handle/hold the butter to put it in/on the pot),
  and no impossible actions or errors occurred.
"}{"
  idx":0,"
  evaluation":"

```

Figure 6: GPT-5-Alignment-response-bug.

```

In [ ]: """
A set of 'key': values is an object; all objects must be placed in
generating multiple independent objects is prohibited.
output ONLY one JSON array:
{
  "idx":0,
  "evaluation":"yes|no",
  "short_justification":"one sentence"
}
"""

```

Figure 7: GPT5Alignmentpromptincrease.

D GPT-5 Experiment Results and Discussion

Similarly, we keep **GPT-5** constant to evaluate all generated games in ByteSized32Refactored and the original ByteSized32. While we were able to obtain valid game generations from GPT-5, we encountered challenges during the evaluation and reflection process, as GPT-5 was unable to execute the provided evaluation and reflection code and prompt successfully. It is worth that rather than indicating a weakness of the RefactoredByteSized32 corpus, the incompatibility reflects GPT-5's different API constraints (e.g., output formatting, limited batch generation, timeout issues) and underscores the need for more robust, model-agnostic evaluation pipelines, the failure and experiment process for GPT-5 which highlights the practical limitations

of our current evaluation methods when applied to LLMs and to suppr futher development at the juncture of world modeling and code generation.

Challenge 1:Output Format GPT-5 often deviated from the expected output format required by the evaluation scripts, leading to parsing errors and failed executions during the *Physical Reality Alignment* evaluation metrics. Despite prompt adjustments shown in new add Prompt 7, GPT-5 failed to produce outputs in the expected fixed JSON format with syntax errors and logical inconsistencies resulting in Figure 6. The error include **JSON Structure Errors**, such as missing or mismatched brackets (e.g., a “” without a corresponding “”), rendering the output unparseable; **Syntax Irregularities**, such as concatenated words (e.g., "theevaluation") without necessary spaces, causing readability and parsing issues; and **Structural Disarray**, where outputs exhibited syntactic and semantic incoherence, with poor sentence transitions, broken logical relationships, and contradictory semantics, resulting in outputs that lacked readability and consistency.

Challenge 2: Timeouts and API Maximum Limitations. While we successfully used GPT-5 to generate all games, the reflection stage proved problematic. For several games (e.g., [specific games]), the multi-iteration reflection process exceeded the preset 30-minute timeout, and GPT-5's higher latency further disrupted the iterative improvement cycle. To proceed with later stages, these games had to be removed from the directory to continue evaluation and reflection. In addition, GPT-5's API enforced a strict maximum call limit of $n \leq 8$, which prevented us from executing the compliance-majority-vote procedure that requires 31 generations.

E Discussion-LLMs Difference: Do Newer LLMs Surpass GPT-4 in World Modeling?

In addition to examining the impact of code structure through our Refactored vs. Original comparison, we also investigated a complementary question: do newer LLMs surpass GPT-4 in world modeling when evaluated on the same original ByteSized32 corpus? While our main focus is on the structural effects introduced by ByteSized32Refactored, this second line of analysis provides insight into model-level differences independent of corpus design. For clarity, we present the results of this comparison in a separate subsection (and provide extended details in the Appendix).

E.1 GPT-4o vs GPT-4 on original Bytesized32

We keep the original Bytesized32 in constant and evaluate all generated games ($N = 96$) in GPT-4o and GPT-4, reporting results both before and after self-reflection to investigate whether the newer GPT-4o surpasses GPT-4 in building world models without Bytesized32Refactored.

The results of the *technical validity* are summarized in Table 4 and Table 7.

Table 7: Technical validity of GPT-4-32k on **ByteSized32Refactored** across reflection steps (0–3).

Technical Validity Measurement	Number of Reflections			
	0	1	2	3
Game Initialization	85.4%	85.4%	89.6%	88.5%
Valid Actions	80.2%	83.3%	87.5%	88.5%
Runnable Game	28.1%	42.7%	51.0%	57.3%

On technical validity, GPT-4o consistently outperforms GPT-4 across all three metrics on the original ByteSized32 corpus. On game initialization, GPT-4o achieves 91.67% valid implementations before reflection, increasing to 95.83% after three reflections. In contrast, GPT-4 starts lower at 85.4% and rises to 89.6%, showing a +4.2% improvement but still trailing GPT-4o by a significant margin of 4.16%. On valid actions, GPT-4o begins at 75.00% and increases to 90.62%, while GPT-4 starts at a lower 80.2% and improves to 88.5%, resulting in a substantial gap of 8.3% after reflection. Finally, on runnable games, GPT-4o starts at 48.96% and rises to 82.29%, whereas GPT-4 begins at a much lower 28.1% and only reaches 57.3%, leaving a gap of 29.2% after reflection.

Similarly to the results of the *physical reality alignment*, *specification compliance* and *winnability of games* are summarized in Table 6, Table 8, Figure 4 and Figure 8. GPT-4o improves from 9.5%

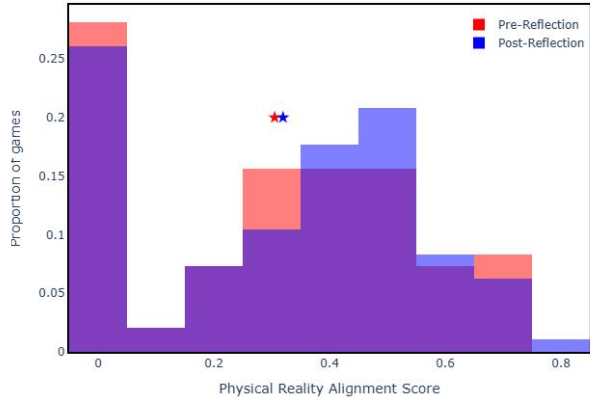


Figure 8: Histogram of physical reality alignment scores before and after reflection.

to 14.8% (+5.3%), while GPT-4 rises only from 30.25% to 31.97% (+1.72%). Score distributions show GPT-4o produces more mid- to high-range results, whereas GPT-4 clusters at lower scores. This indicates GPT-4o benefits more from reflection and achieves a more balanced alignment.

These results indicate that GPT-4o has a

Table 8: Specification compliance and winnability for GPT-4-32k on **Original ByteSized32** before/after reflection.

Measurement	Reflection		Δ
	Before	After	
<i>Specification Compliance</i>			
Task-critical objects	100.0%	100.0%	0.0%
Task-critical actions	93.8%	93.8%	0.0%
Distractors	21.9%	18.8%	-3.1%
Winnability	30.2%	37.5%	+7.3%

stronger initial capability and benefits more from self-reflection compared to GPT-4. GPT-4-32K achieves a higher initial alignment score (0.3025 vs. 0.095) but gains little from reflection (+0.0172), whereas GPT-4o starts much lower but improves more substantially (+0.053). This indicates that GPT-4-32K’s generations are already physically grounded, yet its reasoning benefits less from iterative correction, while GPT-4o, despite its weaker initial grounding, shows stronger reflection-driven reasoning.

F **RefactoredBYTESIZED32 PYTHON
TEMPLATE in GPT-4o**

GPT Game Generation Prompt

You are DeveloperGPT, the most advanced AI developer tool on the planet. You answer any coding question, and provide real useful example code using code blocks. Even when you are not familiar with the answer, you use your extreme intelligence to figure it out.

Your task is to write a program that: is a text-based simulation. The program should be written in Python. It should be challenging to the user, testing their common-sense knowledge, and take multiple steps to complete. If possible, there should be distractor objects and actions that do not help progress, to measure whether the user really knows what they're doing. You should name all target objects and distractor objects with common-sense names.

```
'''python GameBasic.py\n'''  
{GameBasic_CODE}  
'''
```

GameBasic.py includes the classes GameObject, Container, Device, Substance, World, Agent, and a base implementation of TextGame. In your game code, you must derive new classes from these basic classes to build your game logic. Your code must contain a class named TextGame. The TextGame class should have the following member functions:

Your code must contain a derived class of TextGame that has the following member functions: `__init__(self, randomSeed)`, `getTaskDescription(self)`, `generatePossibleActions(self)`, `step(self, actionStr)`, `calculateScore(self)`.

Here is a specification of the task that your code should simulate.

Task: Create a micro-simulation that models how to heat milk to a temperature that is suitable for a baby using a stove.

Environment: kitchen

Task-critical Objects: Stove, Pot, Milk, Fridge, Thermometer

High-level object classes: Device (Stove, Fridge), Container (Stove, Pot, Fridge)

Critical properties: temperature (Milk), temperature_increase_per_tick (Stove), temperature_decrease_per_tick (fridge), max_temperature (Stove), min_temperature (fridge)

Actions: look, inventory, examine, take/put object, open/close container, turn on/off device, use thermometer on object, feed baby with milk

Distractor Items: None

Distractor Actions: drink milk

High-level solution procedure: open fridge, take pot containing milk, put the pot on the stove, turn on the stove, use the thermometer to monitor the milk temperature till the temperature is suitable for a baby to drink, feed baby

```
'''
```

GPT Specification Compliance Prompt

You are DeveloperGPT, the most advanced AI developer tool on the planet. You answer any coding question, and provide real useful example code using code blocks. Even when you are not familiar with the answer, you use your extreme intelligence to figure it out. Your task is to evaluate a program that is a text-based Simulation.

Here is a specification of the simulation: {GAME_SPEC}

Here is the GameBasic library used by the simulation: {GameBasic_CODE}

Here is the code of the simulation: {GAME_CODE}

Answer the following question based on the given specification and the simulation code: {EVAL_QUESTION}

Answer 'Yes' or 'No' first and briefly explain your answer.

GPT Physical Reality Alignment Prompt

in the playthrough of the text game below, I would like you to describe whether the game engine (i.e. the observations it returns in response to actions) are physically accurate models of the world or whether they don't make sense.

An example of not making sense would be being able to take an action from a container (like a fridge) without having opened it first. In addition, if an action produces an error from the game, then it automatically fails to accurately model the world and does not make sense.

Please restrict your evaluation only to the short playthrough, and the specific actions chosen, without speculating about other Actions.

Note: Objects can be manipulated by the agent without first being explicitly picked up, as long as they are in the environment, and readily accessible (e.g. not in a closed container).

The evaluation should be binary ("yes" or "no"), except in the cases where the code generated an error, when the evaluation should be "error".

Here is an example output format: {"evaluation": "no", "short_justification": "could take an object (banana) from the closed fridge without having to first open the fridge"}

Game Task: {GAME_TASK}

Here is the playthrough to evaluate: {PATH}

GPT Reflection Prompt

You are DeveloperGPT, the most advanced AI developer tool on the planet. You answer any coding question, and provide real useful example code using code blocks. Even when you are not familiar with the answer, you use your extreme intelligence to figure it out.

Your task is to correct a program that is a text-based simulation. Here is the code of the simulation

```
'''  
{GAME_CODE}  
'''
```

Here is the error message from a Python interpreter.

```
{ERROR_MESSAGE}
```

You should respond all the code with your fix. Do not respond anything else.

GPT Winnability Prompt

You are playing a text-based games. Interact with the environment to solve a task.

Here is an example.
{IN_CONTEXT_EXAMPLE}

The game you are about to play only understands one command at a time from the following list of commands:
{POSSIBLE_ACTIONS}

Prepend your thoughts with 'think:' when planning your next steps. When you think the task is completed, say 'done'. If you believe the task truly cannot be completed, or if you are repeating failed actions with no progress, or if all actions only result in errors, then say 'bug'.

The IN_CONTEXT_EXAMPLE used in the GPT Winnability Prompt

Task Description: Your task is to light the LED by building a circuit.

Initial Observation: You find yourself in a workshop. In the workshop, you see:

- yourself
- a LED which is off
- a red wire
- a black wire
- a blue wire
- a battery

Type 'help' for a list of possible actions.

> think: To light the LED, I need to create a complete circuit. This will involve connecting the LED to the battery using the wires. I will need to connect one terminal of the LED to the positive terminal (anode) of the battery and the other terminal of the LED to the negative terminal (cathode) of the battery. I will use the red wire for the positive connection and the black wire for the negative connection. The blue wire may not be necessary, but I will keep it in mind in case it is needed. I will start by examining the items in the workshop.

> look around

Observation: You find yourself in a workshop. In the workshop, you see:

- yourself
- a LED which is off
- a red wire
- a black wire
- a blue wire
- a battery

> think: I need to gather the necessary components to build the circuit. I will start by taking the LED, the red wire, the black wire, and the battery into my inventory.

> take LED

The LED is removed from the workshop. You put the LED in your inventory.

> think: I will now take the battery into my inventory, as it is the final necessary component for building the circuit.

> take battery

The battery is removed from the workshop. You put the battery in your inventory.

> think: I now have all the necessary components in my inventory: the LED, the red wire, the black wire, and the battery. I will proceed to build the circuit by connecting the components. First, I will connect one terminal of the LED to the red wire, which will later connect to the positive terminal (anode) of the battery.

> connect LED terminal1 to red wire terminal1

Successfully connected LED terminal1 to red wire terminal1

> think: I have successfully connected one terminal of the LED to the red wire. Next, I will connect the other terminal of the LED to the black wire, which will later connect to the negative terminal (cathode) of the battery.

> connect LED terminal2 to black wire terminal1

Successfully connected LED terminal2 to black wire terminal1

> think: I have successfully connected the LED to both the red wire and the black wire. Now, I need to complete the circuit by connecting the red wire to the positive terminal (anode) of the battery and the black wire to the negative terminal (cathode) of the battery. I will start by connecting the red wire to the battery's anode.

> connect red wire terminal2 to battery anode

Successfully connected red wire terminal2 to battery anode

> think: I have successfully connected the red wire to the battery's anode. Now, I need to complete the circuit by connecting the black wire to the battery's cathode. Once this connection is made, the circuit should be complete, and the LED should light up.

> connect black wire terminal2 to battery cathode

Successfully connected black wire terminal2 to battery cathode

Current step: 7

Score: 1

Reward: 1

Game Over: True

Game Won: True