Distilling Game Code World Model Generation into Lightweight Large Language Models

Tyrone Serapio
Brown University
Providence, United States
tyrone_kirk_serapio@brown.edu

Arjun Prakash
Brown University
Providence, United States
arjun_prakash@brown.edu

Haoyang Xu
Brown University
Providence, United States
haoyang_xu@brown.edu

Amy Greenwald
Brown University
Providence, United States
amy@brown.edu

ABSTRACT

Large Language Models (LLMs) have shown great ability in generating executable code from natural language, opening the possibility of automatically constructing environments for AI agents. Recent work on Code World Models (CWMs) demonstrates that LLMs can translate game rules into Python implementations compatible with solvers like Monte Carlo Tree Search. We study this problem in game settings, where generated environments must implement rules, legal actions, state transitions, observations, and rewards. We refer to these game-specific executable models as Game Code World Models (GameCWMs). However, current approaches to generating code world models rely on frontier models and inference-time refinement loops, limiting accessibility and scalability. This work investigates whether GameCWM generation capabilities can be distilled into smaller models through post-training. We introduce: (1) a curated dataset of 30 games spanning perfect and imperfect information games, (2) a verification framework that evaluates generated code against structural and semantic game properties, and (3) a post-training pipeline combining Supervised Fine-Tuning (SFT) with Reinforcement Learning with Verifiable Rewards (RLVR). We experiment with Qwen2.5-3B-Instruct and find that SFT can increase syntactic correctness, while RLVR can improve execution-level adherence to game rules, thereby improving Qwen's ability to generate valid GameCWMs in both perfect and imperfect information games. Overall, our pipeline makes Qwen2.5-3B-Instruct more capable of generating valid GameCWMs, thereby offering a scalable path toward automatic environment generation from natural language.

KEYWORDS

Game Theory, Large Language Models, Reinforcement Learning

ACM Reference Format:
Tyrone Serapio, Arjun Prakash, Haoyang Xu, and Amy Greenwald. 2026. Distilling Game Code World Model Generation into Lightweight Large Language Models. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 16 pages.

Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus. © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). This work is licenced under the Creative Commons Attribution 4.0 International (CC-BY 4.0) licence.

1 INTRODUCTION

A world model is an AI's internal representation of the world with which it is designed to interact. It is a description of how the AI's environment works, which enables the AI to reason before it acts by simulating the effects of its actions internally. While world models are immensely valuable once deployed, building them can be an onerous task. As a result, recent research has been geared towards automating the building process.

One straightforward method is to assume that LLMs are sufficiently capable to serve as world models themselves. With this approach, you simply ask the LLM to weigh the consequences of its future actions before taking any. This approach has not been particularly successful to date, especially in long-horizon tasks, as LLMs tend to generate unworkable plans [20]. Moreover, their performance degrades in out-of-distribution environments [3].

An alternative is to give an LLM a natural-language description of an environment, and ask it to generate code that faithfully models the physical world as described [7, 18, 31]. As long as we can verify the faithfulness of the world model to the given environment, this latter approach should prove more reliable and interpretable [6]. To this end, Dainese et al. [6] introduced the concept of Code World Models (CWMs) to construct executable Python environments for single-agent reinforcement learning environments.

Today's world models are primarily designed for a single agent, and even when additional agents are present in the environment, all but one are usually assumed to be passive: e.g., in Mujoco [28].[1] Building on Dainese et al.'s work, Lehrach et al. [18] extended CWMs to multi-agent game-theoretic environments to fill this gap. Just like for single-agent settings, the goal is to sidestep the task of manually coding a game environment, and to instead input a game description into an LLM and prompt it for a model of the game as code for a standard library of game solvers, such as OpenSpiel [16]. We refer to Lehrach et al.'s CWMs for games as Game Code World Models, or GameCWMs for short.

Using LLMs for GameCWM-generation is a promising direction for streamlining their creation for novel multiagent environments. Current approaches, however, face significant accessibility and scalability barriers. For example, only frontier models (e.g., Gemini 2.5 Pro) have achieved acceptable success rates thus far [18]. Furthermore, Lehrach et al. [18], Tang et al. [26], Dainese et al. [6],

[1] One notable exception is multi-robot model predictive controllers [25].

and Deng et al. [7] all rely on extensive test-time compute (i.e. computational effort exerted during inference) to search for, verify, and refine generated code. More specifically, they use iterative-refinement loops, in which an LLM generates code, that is then checked—against a provided suite of unit tests—after which the LLM is called upon to try again, as needed. Our goal in this work is to tackle these barriers. That is, our aim is to make the automated generation of syntactically and semantically correct GameCWMs more resource efficient.

The approach we take is a natural one. We attempt to distill GameCWM generation into smaller models through post-training, i.e., refining a pre-trained base model through supervised and/or reinforcement learning [13]. We hypothesize that reasoning about complex tasks, currently the burden of test-time compute, can be incorporated into post-training, whereby lightweight models would internalize capabilities that were previously available only in frontier LLMs.

We introduce a pipeline (see Figure 1) to post-train Qwen2.5-3B-Instruct [27] to generate GameCWMs for both perfect and imperfect information games. Learning from a curated dataset of OpenSpiel-like implementations, our approach involves a Supervised Fine-Tuning (SFT) stage followed by a Reinforcement Learning with Verifiable Rewards (RLVR) stage [15].[2] In the latter, we use Group Relative Policy Optimization (GRPO) [23], and we build an execution-based verifier that automatically assigns reward signals based on game-theoretic properties.

The specific contributions of this work are the following:

(1) **Dataset Curation:** We introduce a comprehensive dataset of extensive-form games implemented via an OpenSpiel-like API. The games were selected to test different axes of LLMs' abilities to generate GameCWMs: mechanical state-tracking (primarily through perfect information games), imperfect information handling, and generalization to novel games.
(2) **Verification Framework:** Our framework validates LLM-generated GameCWMs with respect to 1. Python syntax; 2. structural game-theoretic properties; and 3. semantic game rule adherence. To verify structural properties, we develop a property-based testing suite that evaluates GameCWMs, without relying on oracle trajectories sourced from pre-existing game engines (e.g., OpenSpiel). To verify semantic adherence, we build an execution-based verifier that evaluates game trajectories produced by the CWM by comparing them to trajectories produced directly from the game rules by a *frontier* LLM. This approach allows us to evaluate LLM-generated GameCWMs even for novel games for which simulated game data is not available.
(3) **Distilling GameCWM generation through Post-Training:** We show that post training (SFT + RLVR) can be used to distill game APIs and game logic into small open-weight LLMs, and thereby improve their generation of valid GameCWMs in both perfect and imperfect information games.

While we are able to demonstrate the potential of post-training for GameCWM distillation in Qwen2.5-3B-Instruct (i.e., an LLM at the 3B parameter scale), as SFT and RLVR do improve its performance, there are limitations to our approach. In particular, imperfect information games remain challenging. We discuss these limitations and directions for future work in Section 6.

[2]Our code and dataset are available here.

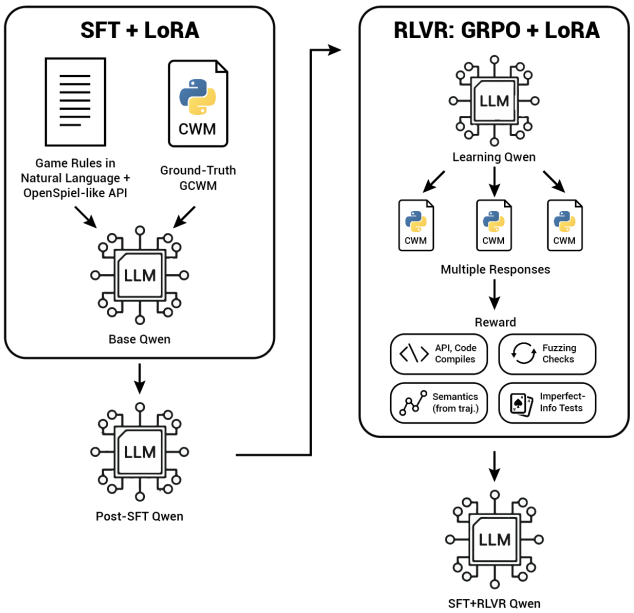

Figure 1: Our Distillation Pipeline. Unlike prior approaches that rely extensively on test-time compute, we aim to distill game-theoretic logic into the model weights through post-training. Like many LLM pipelines, ours involves two stages: Supervised Fine-Tuning (SFT) followed by RLVR using GRPO. To enable our approach, we define a dataset of prompt-code pairs for use during SFT, and a hierarchical verification framework for automatically evaluating generated code, which provides reward signals that reinforce the syntactic correctness and semantic validity of generated GameCWMs during RLVR.

2 RELATED WORK

Recent literature has explored the intersection of LLMs and game-theoretic environments. This section reviews previous work across: (1) the use of LLMs as strategic game-playing agents, (2) the emerging paradigm of leveraging LLMs to generate environment representations, and (3) the application of reinforcement learning to improve LLM code generation, which serves as the foundation for our post-training pipeline.

2.1 LLMs as Game-Playing Agents

Perhaps the most natural approach is to treat the LLM as a player of the game, and simply prompt it to do some "internal planning" (i.e., to rollout game trajectories in text). This approach, which assumes the LLM is equipped with a latent function over game continuations, has been explored for a diverse set of games, such as Chess [9] and Werewolf [34]. Diplomacy, however, relied on an external strategic reasoning engine to guide the LLM [1].

Although LLMs can achieve reasonable performance via internal planning in games that are well-represented in their training data, they face fundamental limitations when attempting to play complex [22] and novel games [2]. To ground LLMs to the underlying game structure, Gemp et al. [10] create a binding from natural language to the symbolic logic of games, which enables their game play to be externally steered by game solvers.

There is also increasing interest in investigating LLMs in multi-agent settings beyond the formal structure of games. Park et al. [19] introduced Generative Agents, an architecture that leverages LLMs to simulate believable social interactions in a "sandbox" society, and Zhou et al. [37] introduced Sotopia, an open-ended environment designed to characterize the social intelligence of LLMs by simulating interactions in which agents must navigate complex, goal-driven social scenarios. These directions further highlight the need to develop world models for multi-agent scenarios, as such models would facilitate a rigorous study of strategic social interaction.

2.2 LLMs for Game and Environment Generation

Rather than using LLMs as players, an emerging direction aims to use LLMs to generate formal representations of game mechanics from natural language descriptions. GG-Bench [30] is such a pipeline. It works by first generating games, then converting those games into Gymnasium-based [29] environments, and finally training RL agents in those environments. These RL agents can then be used to evaluate other game-playing agents, including LLMs themselves.

Deng et al. [7] use LLMs to construct extensive-form game (EFG) tree representations in `pygambit` from natural language descriptions. Like us, they find that it is much more challenging to generate environments that correctly represent imperfect information games than perfect information games. To tackle this problem, they introduce a two-stage framework: first, an LLM identifies information sets and partial tree structures; second, an LLM generates the EFG code. The generation step utilizes an automated self-debugging loop where the generated code is executed, and then any Python error messages are fed back into the LLM for correction. Manual checks are performed afterwards to ensure consistency with the game rules.

Dainese et al. [6] explore the generation of Code World Models (CWMs) for model-based RL by leveraging LLMs to write executable Python representations of RL environments given natural language descriptions. To address the challenges of generating accurate transition and reward logic, they introduce Generate, Improve and Fix with Monte Carlo Tree Search (GIF-MCTS), an inference-time code generation strategy. GIF-MCTS guides the LLM to iteratively generate, self-debug, and refine candidate programs using feedback based on ground-truth environment trajectories. While highly effective, this methodology relies heavily on test-time compute to successfully search for and refine the generated code. Moreover, this work is limited to single-agent, perfect-information environments.

Our work is inspired by Lehrach et al. [18], who used GameCWMs for multi-agent general game playing, including imperfect-information game environments with partial observability. Similar to the approaches of Dainese et al. [6] and Deng et al. [7], Lehrach et al.'s method relies on an iterative refinement loop, where a frontier LLM generates code, tests it using a suite of unit tests and ground-truth game trajectories, and regenerates code upon failure. While effective, these methodologies [6, 7, 18] require substantial test-time compute and frontier models (e.g. GPT-4, Gemini 2.5 Pro) to achieve acceptable success rates. We aim to distill GameCWM generation into smaller models via post-training, ideally eliminating dependency on iterative refinement at test time and on frontier models.

2.3 Reinforcement Learning for Code Generation

Reinforcement learning (RL) has emerged as a critical post-training technique for improving code generation capabilities beyond what SFT alone can achieve [17]. More specifically, we use RLVR [14, 32] to generate a reward signal by verifying code output with test cases. The credibility of this approach was demonstrated by DeepSeek-R1 [11]. We also draw inspiration from hierarchical approaches such as StepCoder [8], which breaks down complex coding challenges into sub-tasks, a technique we mimic in our proposed hierarchical verification framework.

Among the different approaches to RL for LLM-based code generation, Group Relative Policy Optimization (GRPO) [23] has become a preferred choice, as compared to Proximal Policy Optimization (PPO) [21], because verifiable rewards are clean signals, and as such are sufficient for computing the relative advantage of one candidate's reward as compared to that of the group.

3 BACKGROUND

In this section, we define extensive-form games, which is the class of games of interest in this paper. Extensive-form games are dynamic games that are represented as trees, which is an exponentially more compact representation than the classic normal form.

3.1 Extensive-Form Games

Extensive-form games (EFGs) are often used to model strategic interactions in multi-player sequential games [24]. Such games are typically modeled by game trees, where each state of the game is represented by a node in the tree, and players' actions are represented by edges from one node to another. Non-terminal nodes, where exactly one player moves, are decorated with the set of actions available to the player whose turn it is to move. Terminal nodes, at which there are no further actions, are decorated with payoffs (e.g., win, lose, or draw in zero-sum games). There is typically a chance player as well, who models stochasticity in the environment (e.g., card deals or dice rolls).

In some games, like Chess, all information is available to all players, while in others, like Poker, some information is hidden (e.g., cards). The latter are called **imperfect-information** games. In these games, the players' decision nodes are not individual states, but rather **information sets**, each of which is a set of states that the player to move cannot distinguish from one another.

We define a history h as a sequence of states and actions from the start of the game. Reasoning about how to play at a given information set in imperfect-information games requires that the

player be able to reconstruct the possible histories that could have led her to that information set.

3.2 Game Code World Models (GameCWMs)

We define a GameCWM as an executable program generated by an LLM that implements the rules of a game environment as executable functions.

A GameCWM provides functions that act as the engine for a game, including state transition dynamics, legal actions, and observations for each player. We implement this interface in Python, using the following methods:

- `apply_action`: Returns the successor state after applying an action.
- `get_legal_actions`: Enumerates valid actions from a given state.
- `get_current_player`: Identifies which player acts (or indicates terminal/chance nodes).
- `get_rewards`: Returns payoffs for each player at terminal states.
- `get_observations`: Returns player-specific observations.

Additionally, for imperfect-information games, we also require a `resample_history` function which, given a player's current information set, samples a complete history—i.e., a sequence of states and actions—consistent with her observations.

A key feature of GameCWMs is their compatibility with game solvers. Once a syntactically correct GameCWM is generated, Monte Carlo Tree Search (MCTS) [4] or Information-Set MCTS (ISMCTS) [5] can be applied immediately without additional training—assuming a semantically correct GameCWM—enabling strong game play from a natural language specification alone.

4 METHODOLOGY

Our approach comprises three components: (1) a curated dataset of 30 games spanning varying difficulty levels, (2) a hierarchical verification framework that evaluates the generated GameCWMs, and (3) a two-stage post-training pipeline combining Supervised Fine-Tuning (SFT) with Group Relative Policy Optimization (GRPO).

4.1 Dataset Construction

We construct a dataset of prompt-code pairs (P_G, C_G), for every game G, where the input prompts P_G contain:

(1) The API specification outlining the required GameCWM functions.
(2) The game description expressed in natural language.
(3) Explicit names of allowed action strings. By establishing naming conventions, we seek to avoid syntactic action-space mismatches, thereby enabling the verification framework to reward semantic adherence to the game rules.
(4) An example trajectory, or **scenario trace**, which is a natural language playthrough generated by a frontier LLM, simulating a human reporting their step-by-step experience traversing the environment.

The target output C_G contains the ground-truth Python code, including complete function implementations, imports, and type definitions and annotations. We present our prompts in Appendix A; they are adapted from Lehrach et al. [18].

We categorize games along the following axes, which guide the Supervised Fine-Tuning stage:

- **Easy Games:** Basic perfect information games with simple state spaces (e.g. Tic-Tac-Toe, Connect4).
- **Intermediate Games:** More complex perfect information games (e.g. Y, Breakthrough) and slightly easier imperfect information games (e.g. Blackjack, Kuhn Poker).
- **Hard Games:** More complex imperfect information games (e.g. Leduc Poker, Poker Hold'em).
- **Novel / Out-of-Distribution (OOD) Games:** Perfect and imperfect information games that are not present in standard LLM training data (Hand of War, Quadranto, Generalized Tic-Tac-Toe, Generalized Chess [18]). In this paper, we also introduce a new OOD game, Converge, created by the authors and Gemini 3 Pro.

To construct ground-truth implementations, we prompt Gemini 3 Pro to generate candidate ground-truth GameCWMs based on game rules, and iteratively refine these models using a loop similar to that of Lehrach et al. [18]. Each implementation must pass a comprehensive test suite: for standard and pre-existing games, we adapt tests from OpenSpiel [16] and also use our verification framework; for novel OOD games, we write custom tests, and again use our verification framework as well. This process is designed to ensure high-quality training data, while avoiding the computational cost of iterative refinement at test time.

4.2 Verification Framework

A core contribution of this work is a verification framework that evaluates GameCWMs by combining property-based testing and lightweight game trajectory matching. We say that our form of trajectory matching is lightweight because rather than using pre-existing game engines to generate ground-truth game trajectories, we test against a small set of game traces created by the inputting the game rules to an LLM. This design offers two advantages: (1) it enables rigorous evaluation even of novel games for which ground-truth data do not exist, and (2) it provides a fast, sparse reward signal for RL training. The framework comprises four tiers. We provide the pseudocode for the implementation of each tier of the evaluation framework in Appendix A.

Tier 1: Static Analysis. First, we verify syntactical and structural validity:

- API Completeness: The class must implement all required methods listed in the prompt.
- Type signatures are repsected (e.g., `get_legal_actions` must return a list).
- All required libraries are imported.
- Code compiles cleanly.

Tier 2: Dynamics (Fuzzing). We fuzz the generated GameCWM with 100 random synthetic trajectories to ensure core physics engine invariants hold:

- Determinism: Applying action a to state s must always yield the identical state s' (excluding chance nodes).

- Non-terminal Logic: If a state is non-terminal, `get_legal_actions` must return a non-empty list.
- Terminal Logic: If a state is terminal, `get_legal_actions` must return an empty list.
- Crashes in Execution: The game must not raise exceptions (e.g., `KeyError`).
- State Immutability: `apply_action` must return a new state object.

Tier 3: Semantic Rule Adherence (LLM-generated Scenario Traces). To verify that the generated engine adheres to the rules of the game, we evaluate the generated GameCWMs against a small set of LLM traces we call **scenarios** derived *a priori* from the game rules. Examples appear in Appendix B.

- For each game, we prompt a frontier model with a game description, and ask it generate a sparse, representative suite of roughly five scenarios for that game, each of which is a history of gameplay.
- These scenarios test boundary conditions (e.g. player win states, maximum-turn draw limits, rejection of illegal actions) and typical gameplay.
- The reward signal for this tier is continuous: the candidate GameCWM receives a score proportional to the fraction of scenarios it passes successfully.

Tier 4: Information Consistency. In imperfect-information games, players do not have access to full state information. Following Lehrach et al. [18], we evaluate the `resample_history` method to ensure epistemic boundaries are maintained.

- We simulate a random walk from the initial state to a valid state s, extract player p's observation O_p at s.
- We then prompt `resample_history` to generate a hypothetical history h' to state s.
- Finally, we execute h' from the initial state, and assert whether the resulting observations, O'_p and O_p, differ. If they do, the model has failed.

Although this tier tests observation consistency across resampled histories, it does not verify against illegal information leakage (e.g. ensuring the underlying `state` dictionary the LLM created within the GameCWM perfectly isolates hidden variables from the observation functions), which we will integrate in future work.

We also design reward computation to be gated: tier 1 must pass before dynamics are evaluated, and dynamics must score ≥ 0.5 before tiers 3 and 4 contribute. In addition, Qwen seems to recognize that `resample_history` is challenging to implement; it leaves stubs in the GameCWM (i.e., returning an empty list). As a result, we added a check for these stubs, and an accompanying penalty during RLVR to tier 4 of our verification framework.

4.3 Training Pipeline

Supervised Fine-Tuning (SFT). We initialize from Qwen2.5-3B-Instruct [27] and fine-tune on the (P_G, C_G) pairs. To support reproducibility, we specify our training hyperparameters and train/eval split in Appendix C and Appendix D, respectively.

RLVR via Group Relative Policy Optimization (GRPO). Following a SFT phase, we apply GRPO as developed by Shao et al. [23] to further align the model with our hierarchical verification framework, thereby executing RLVR.

To generate candidate GameCWMs during the RLVR stage (as well as during our evaluations), we prompt the model with P_G for a given game G, and consider its zero-shot output. For each generated GameCWM, we then compute a composite reward:

$$r_i = \sum_{t=1}^{4} w_t S_t$$

where $S_t \in [0, 1]$ is the verification score at tier t, meaning the proportion of tests that pass at tier t). We set $w_1 = 0.15$ (static), $w_2 = 0.25$ (dynamics), $w_3 = 0.3$ (semantics), and $w_4 = 0.3$ (information handling), weighing semantics and information handling more heavily to prioritize semantic game logic over mere API compliance.

5 EXPERIMENTS

We evaluate the extent to which post-training with SFT, GRPO, and SFT+RLVR enables the distillation of GameCWM generation capabilities into small models, comparing the contributions of SFT and RLVR both independently and jointly.

5.1 Experimental Setup

Training. We train four models built on top of Qwen2.5-3B-Instruct [27] on 23 of the 30 games in our data set. We use the remaining 7 as a held-out test set. The four models are:

- **Base**: The original Qwen2.5-3B-Instruct before applying our post-training pipeline.
- **SFT**: Fine-tuned on our dataset using supervised learning using next-token prediction loss with LoRA [12].
- **RLVR**: Base model trained directly with GRPO using our execution-based verification as a reward signal.
- **SFT+RLVR** (our full pipeline): SFT model further refined with GRPO.

Evaluation. We evaluate each model on a set of 7 held-out games, which includes novel/OOD and non-novel (in-distribution) perfect- and imperfect-information games.[3] The perfect-information games are Generalized Tic-Tac-Toe (OOD), Generalized Chess (OOD), Converge (OOD), Y; the imperfect-information games are Quadranto (OOD), Hand-of-War (OOD), and Gin Rummy. This setup is summarized in Table 1. For each, we generate 25 samples per game, and report average verification scores across the four verification tiers.[4]

Table 1: Categorization of held-out games based on information type and distribution.

	Perfect Information	Imperfect Information
In-Distribution	Y	Gin Rummy
Out-of-Distribution (OOD)	Gen. Tic-Tac-Toe, Gen. Chess, Converge	Quadranto, Hand of War

[3]We consider the games introduced by Lehrach et al. [18] to be novel/OOD with Qwen2.5-3B-Instruct because the Gemini model they use was published after the release date of the Qwen2.5 models.
[4]We set Qwen's temperature to be 0.3, to balance accuracy and flexibilty.

Table 2: Overall mean verification score with 90% and 95% confidence intervals across held-out games. Intervals are computed over 25 generations per game. SFT+RLVR yields the strongest Qwen-based model. Note the statistically significant separation between it and the base model.

Model	Mean	90% CI	95% CI	N
Base	47.3%	[44.8, 49.8]%	[44.3, 50.3]%	175
SFT	49.0%	[46.8, 51.2]%	[46.4, 51.6]%	175
RLVR	50.4%	[48.6, 52.1]%	[48.3, 52.4]%	175
SFT+RLVR	53.2%	[51.3, 55.1]%	[51.0, 55.4]%	175
GPT-4o	66.7%	[64.5, 69.0]%	[64.1, 69.4]%	175

Table 3: Per-tier average verification scores across all held-out games. Values are averaged across 25 sample generations per game. Information is averaged only over imperfect-information games. SFT+RLVR yields the strongest performance in all categories among the post-trained models.

Model	Static	Fuzz	Semantics	Information
Base	77.5%	70.2%	8.5%	3.3%
SFT	78.0%	74.3%	9.4%	2.0%
RLVR	83.3%	75.3%	8.4%	3.7%
SFT+RLVR	86.3%	75.1%	14.4%	5.3%
GPT-4o	98.4%	84.6%	33.8%	14.0%

Note: A significant number of the outputs by the base Qwen model do not compile at first, simply because of missing imports. To ensure that we are testing Qwen's ability to generate valid GameCWMs, rather than its ability as a Python software engineer, we inject these missing imports via a preprocessing step.

5.2 Results

Table 2 reports mean verification scores across all held-out games and all verification tiers for various confidence levels. Table 3 disaggregates these rates by tier, and Table 4 further disaggregates by game. Tier 1 has seven tests, Tier 2 has four tests, Tier 3 has five tests, and Tier 4 has four tests, with each tier's tests capturing the components discussed in Section 4.2. A score of 0.0% indicates that no generated GameCWM passed any test in that tier's verification suite. A few generated GameCWMs of interest can be found in Appendix F; the rest are published in our GitHub repository.

Based on our experiments, several findings emerge.

Overall, SFT+RLVR post-training yields the strongest Qwen-based model. As shown in Table 2, SFT+RLVR produces the most capable Qwen model in our study, demonstrating a statistically significant separation over the base model's performance. Despite these improvements, GPT-4o remains significantly stronger overall.

SFT improves syntactical and structural correctness. As shown in Table 3, SFT can improve Syntax (Tier 1) and Fuzz (Tier 2) scores compared to the base model. More specifically, in Table 4, we see that SFT increased the Fuzz score (or it is unchanged) in all games. In addition, on most OOD games, SFT increases the Static score as compared to the base model. (An exception is Quadranto, but the decrease was small.)

Table 4: Aggregate verification scores across the four verification tiers. Values are averaged across 25 sample generations per game. Semantics are challenging for all models, including GPT-4o, as is information-handling in imperfect-information games.

		Test Hierarchies			
Game	**Model**	**Static**	**Fuzz**	**Semantics**	**Information**
Perfect Information Games					
Gen. Tic-Tac-Toe (OOD)	Base	90.3%	65.6%	17.7%	N/A
	SFT	98.3%	68.0%	26.3%	N/A
	RLVR	98.9%	65.6%	23.4%	N/A
	SFT+RLVR	96.0%	62.4%	20.6%	N/A
	GPT-4o	100.0%	100.0%	80.0%	N/A
Converge (OOD)	Base	92.0%	78.4%	0.0%	N/A
	SFT	97.7%	82.4%	0.0%	N/A
	RLVR	96.6%	80.0%	1.6%	N/A
	SFT+RLVR	91.4%	82.4%	0.0%	N/A
	GPT-4o	96.0%	91.2%	52.0%	N/A
Gen. Chess (OOD)	Base	61.1%	76.8%	1.6%	N/A
	SFT	64.6%	76.8%	4.8%	N/A
	RLVR	61.7%	73.6%	5.6%	N/A
	SFT+RLVR	74.9%	76.8%	20.8%	N/A
	GPT-4o	93.1%	80.0%	33.6%	N/A
Y	Base	90.9%	75.2%	16.8%	N/A
	SFT	92.0%	78.4%	15.2%	N/A
	RLVR	92.6%	76.8%	15.2%	N/A
	SFT+RLVR	97.7%	72.8%	34.4%	N/A
	GPT-4o	100.0%	80.8%	19.2%	N/A
Imperfect Information Games					
Quadranto (OOD)	Base	84.6%	76.8%	16.8%	6.0%
	SFT	82.3%	82.4%	12.8%	6.0%
	RLVR	81.7%	80.0%	8.0%	5.0%
	SFT+RLVR	86.9%	80.8%	15.2%	6.0%
	GPT-4o	100.0%	88.0%	33.6%	18.0%
Hand of War (OOD)	Base	73.7%	72.8%	5.6%	1.0%
	SFT	79.4%	76.0%	6.4%	0.0%
	RLVR	84.0%	76.8%	4.8%	1.0%
	SFT+RLVR	89.7%	75.2%	9.6%	1.0%
	GPT-4o	100.0%	72.0%	18.4%	19.0%
Gin Rummy	Base	49.7%	45.6%	0.8%	3.0%
	SFT	31.4%	56.0%	0.0%	0.0%
	RLVR	67.4%	74.4%	0.0%	5.0%
	SFT+RLVR	67.4%	75.2%	0.0%	9.0%
	GPT-4o	100.0%	80.0%	0.0%	5.0%

In Gin Rummy, however, an in-distribution game for the base model that we did not include in our SFT training set, SFT decreased performance as compared to the base model from 49.7% to 31.4%! We inspected the generated code samples for SFT on Gin Rummy and uncovered several syntax errors, such as mixing list elements with a list comprehension (e.g. `["Draw stock", "Draw upcard", "Action: " + card for card in state["deck"]]`); see Appendix F). The usual explanation for SFT improving performance is that it better aligns a model's output with specialized downstream outputs. In this case, however, SFT appears to have led to overfitting. The syntax errors demonstrate that the model blindly attempts to fit Gin Rummy's mechanics into the list-comprehension templates it learned during SFT, overriding the more syntactically valid code generation capabilities of the base model.

SFT and RLVR, alone and combined, improves Fuzz scores. As seen in Table 3, SFT, RLVR, and SFT+RLVR achieve Fuzz scores ahead of the base model. This result suggests that SFT and RL both help the base model produce transition functions that are more robust under randomized rollouts, but the gains are smaller than those observed at the Static tier. In several games, generated GameCWMs can compile and expose the required API, while still failing to preserve necessary properties for compatibility with standard game theory solvers like MCTS.

.

SFT combined with RLVR improves adherence to game rule semantics. As seen in Table 3, the Semantics score of GPT-4o, a frontier model, is only 33.4%. This finding indicates that translating natural language games into game trajectories that faithfully model a game via a GameCWM is currently a difficult task.

Among all Qwen models, SFT+RLVR obtains the best Semantics score. This result suggests that RL is most useful when applied after SFT. This finding highlights the value of our two-stage post-training method: SFT teaches the model the expected API format and other structure, and RLVR then provides execution-based feedback that improves adherence to game rules.

Even RLVR alone generally improves Semantic adherence in all games (both perfect- and imperfect-information, and both in-distribution and OOD) on our held-out data set. Not in Quadranto, however. Upon inspecting generated Quadranto GameCWMs from the RLVR model, we found that some implementations failed to update the current-player indicator to the terminal value after the game had ended. It seems our current RLVR reward only allows the model to learn simplified transition patterns.

Complex games, especially complex imperfect-information games, remain especially challenging. Across Quadranto, Hand of War, and Gin Rummy, all Qwen models achieve low Information scores, as does GPT-4o. SFT+RLVR performs best among post-trained Qwen models with an average Information score of 5.3%, compared with 14% for GPT-4o. This finding indicates that the logical complexity of these games exceeds the current capacity of both frontier models and our training pipeline.

LLMs explicitly indicate their limitations. Although they were unable to fully generate the complex logic for `resample_history` most of the time, an interesting behavior observed in the models was the explicit comments indicating the complexity of implementation. For example, in several generated code samples, the SFT+RLVR Qwen model inserted comments stating that the resampling logic in `resample_history` was too difficult, opting for a simplified implementation instead (e.g., returning the last action in the history). An example appears in Appendix F.

Our findings suggest that the models possess a high-level understanding of the API requirements, but lack the requisite reasoning abilities to generate logic in a GameCWM that fully adheres to the game rules (e.g., the `resample_history` function).

Summary. These results point to three main conclusions. First, SFT alone is not enough: it does not improve overall verification performance and can reduce performance on some tiers. Second, RLVR improves some structural and rollout properties, but direct RLVR from the base model is less effective than applying RLVR after SFT, which aligns with LLM post-training practices that use SFT before RLVR [33]. Third, the full SFT+RLVR pipeline exhibits the strongest post-trained Qwen results, especially on syntactical correctness and aggregate semantic adherence, but both our models and GPT-4o and struggle with complex semantic and imperfect-information reasoning.

The limitations of SFT and RLVR. Both SFT and RLVR are known to be limited by the capabilities of the base model. Work by Zhou et al. [36] shows that SFT primarily aligns a model to a desired format or style; it does not teach it complex or novel reasoning skills. Le et al. [17] note that relying solely on a next-token prediction objective penalizes the model for exploring valid solutions that deviate from the ground-truth dataset, severely limiting SFT's reasoning capabilities on novel problems.

RLVR faces similar limitations. Recent work by Yue et al. [35] suggests that the reasoning capabilities of RLVR-trained models are bounded by the base model's sampling distribution; that is, if a correct reasoning path does not exist within the base model's latent space with non-zero probability, RL cannot "invent" it. Given that our base model is a lightweight 3B parameter model, it is possible that the complex "Theory of Mind" logic required for these games is simply absent from its pre-training distribution. Indeed, upon inspecting a few generated code samples for Gin Rummy, we found that the post-trained models just return a constant set of actions or simply the last action in the history, just as the base model does.

6 CONCLUSION AND FUTURE WORK

Game theory serves as a framework in which to understand multi-agent interactions, from simple board games to high-stakes scenarios in politics, economics, and corporate competition [24]. En route to building AI agents that behave effectively and reliably in game-theoretic environments, some have argued that a first step is to endow them with game code world models (GameCWMs), executable environments that enforce game rules and manage environment dynamics. While recent work has proposed the use of Large Language Models (LLMs) to construct GameCWMs from natural language, these approaches still rely on computationally expensive test-time simulations and refinement loops, and multiple calls to frontier LLM APIs. In this work, we questioned whether this reliance on expensive compute could be avoided by shifting the computational burden to post-training a small model.

To this end, we presented a pipeline for distilling GameCWM-generation into lightweight LLMs through post-training, aiming to remove the need for frontier models and inference-time refinement loops. Our approach combines a curated game dataset, a hierarchical verification framework based on game-theoretic properties, and a two-stage SFT+RLVR training pipeline. Experiments demonstrate that SFT alone, RLVR alone, and our pipeline enables Qwen2.5-3B-Instruct to become more capable of generating valid GameCWMs in both perfect and imperfect information games. The core of our approach is the use of execution-based verification as a reward signal. We use game-specific behavioral checks to evaluate structural and semantic properties of game environments, including state immutability, determinism, terminal-state behavior, rule adherence, and observation consistency. In this way, we can

post-train GameCWM generators, thereby reducing reliance on expensive frontier-model repair loops at inference time.

Limitations. Several limitations constrain our current results. First, our dataset, while diverse, is likely insufficient to cover the full complexity of game-theoretic reasoning, as seen in our enhanced models' performance on games like Hand of War, where all Qwen models achieve 0% on imperfect information tests. Second, our evaluation only focuses on verification scores and does not evaluate downstream gameplay performance; a generated GameCWM may pass our tests yet still contain subtle bugs affecting MCTS planning quality, which points to future work to complete the pipeline. Third, a limit to the semantics tier of our verifier is that our semantic tests rely on LLM-generated scenario traces. Even when produced by frontier models, these traces are not guaranteed to be correct or comprehensive.

Future Work. Several directions extend from this work:

- **Improving the verification framework.** Our current verifier checks syntax, selected structural properties of game dynamics, semantic adherence on a small set of scenario traces, and observation consistency for imperfect-information games. However, it does not provide complete verification of the generated game. For example, for imperfect-information games, future verifiers should also check information-set correctness more rigorously, ensuring that observation functions only expose information available to the acting player.
- **Scaling data and model size.** Expanding the dataset to include more imperfect-information games and evaluating larger base models (e.g., 14B) may address the current failures on complex games and imperfect-information handling.
- **End-to-end gameplay evaluation.** Integrating generated GameCWMs with MCTS and ISMCTS, and evaluating actual gameplay performance against reference implementations and/or human players could provide a more well-rounded, complete assessment of GameCWM quality.
- **Iterative self-improvement.** Combining our trained model with iterative refinement over unit tests, trajectories, or even our constructed verification framework may recover the benefits of iterative refinement seen in the experiments of Lehrach et al. [18], Tang et al. [26], and Dainese et al. [6]. In particular, our finding of the models' "awareness" of their own limitations could guide an iterative refinement process.

Overall, our results suggest that GameCWM generation can be partially distilled into lightweight LLMs through post-training. This work takes a step forward towards AI systems that can model and reason about complex strategic interactions in critical domains such as economic forecasting, corporate negotiations, and international relations, areas in which early-stage autonomous agents will surely benefit from incorporating language understanding.

REFERENCES

[1] Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Jacob, Mojtaba Komeili, Karthik Konath, Minae Kwon, Adam Lerer, Mike Lewis, Alexander Miller, Sasha Mitts, Adithya Renduchintala, Stephen Roller, and Markus Zijlstra. 2022. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. *Science* 378 (11 2022). https://doi.org/10.1126/science.ade9097

[2] Bahar Bateni, Benjamin Pratt, and Jim Whitehead. 2025. LLM game rule understanding through out-of-distribution fine-tuning. In *Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Edmonton, Alberta, Canada) *(AIIDE '25)*. AAAI Press, Article 1, 10 pages. https://doi.org/10.1609/aiide.v21i1.36804

[3] Shiqi Chen, Tongyao Zhu, Zian Wang, Jinghan Zhang, Kangrui Wang, Siyang Gao, Teng Xiao, Yee Whye Teh, Junxian He, and Manling Li. 2025. Internalizing World Models via Self-Play Finetuning for Agentic RL. arXiv:2510.15047 [cs.LG] https://arxiv.org/abs/2510.15047

[4] Rémi Coulom. 2006. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *International Conference on Computers and Games*. Springer, 72–83.

[5] Peter I Cowling, Edward J Powley, and Daniel Whitehouse. 2012. Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 120–143.

[6] Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. 2024. Generating Code World Models with Large Language Models Guided by Monte Carlo Tree Search. arXiv:2405.15383 [cs.AI] https://arxiv.org/abs/2405.15383

[7] Shilong Deng, Yongzhao Wang, and Rahul Savani. 2025. From Natural Language to Extensive-Form Game Representations. arXiv:2501.17282 [cs.AI] https://arxiv.org/abs/2501.17282

[8] Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Xuanjing Huang, and Tao Gui. 2024. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback. arXiv:2402.01391 [cs.SE] https://arxiv.org/abs/2402.01391

[9] Xidong Feng, Yicheng Luo, Ziyan Wang, Hongrui Tang, Mengyue Yang, Kun Shao, David Mguni, Yali Du, and Jun Wang. 2023. ChessGPT: Bridging Policy Learning and Language Modeling. arXiv:2306.09200 [cs.LG] https://arxiv.org/abs/2306.09200

[10] Ian Gemp, Roma Patel, Yoram Bachrach, Marc Lanctot, Vibhavari Dasagi, Luke Marris, Georgios Piliouras, Siqi Liu, and Karl Tuyls. 2024. Steering Language Models with Game-Theoretic Solvers. *arXiv preprint arXiv:2402.01704* (2024).

[11] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature* 645, 8081 (Sept. 2025), 633–638. https://doi.org/10.1038/s41586-025-09422-z

[12] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[13] Haitao Jiang, Wenbo Zhang, Jiarui Yao, Hengrui Cai, Sheng Wang, and Rui Song. 2026. Supervised Fine-Tuning versus Reinforcement Learning: A Study of Post-Training Methods for Large Language Models. arXiv:2603.13985 [cs.AI] https://arxiv.org/abs/2603.13985

[14] Xue Jiang, Yihong Dong, Mengyang Liu, Hongyi Deng, Tian Wang, Yongding Tao, Rongyu Cao, Binhua Li, Zhi Jin, Wenpin Jiao, Fei Huang, Yongbin Li, and Ge Li. 2026. CodeRL+: Improving Code Generation via Reinforcement with Execution Semantics Alignment. arXiv:2510.18471 [cs.SE] https://arxiv.org/abs/2510.18471

[15] Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. 2025. Tulu 3: Pushing Frontiers in Open Language Model Post-Training. arXiv:2411.15124 [cs.CL] https://arxiv.org/abs/2411.15124

[16] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. 2019. OpenSpiel: A Framework for Reinforcement Learning in Games. *CoRR* abs/1908.09453 (2019). arXiv:1908.09453 [cs.LG] http://arxiv.org/abs/1908.09453

[17] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG] https://arxiv.org/abs/2207.01780

[18] Wolfgang Lehrach, Daniel Hennes, Miguel Lazaro-Gredilla, Xinghua Lou, Carter Wendelken, Zun Li, Antoine Dedieu, Jordi Grau-Moya, Marc Lanctot, Atil Iscen, John Schultz, Marcus Chiam, Ian Gemp, Piotr Zielinski, Satinder Singh, and Kevin P. Murphy. 2025. Code World Models for General Game Playing. arXiv:2510.04542 [cs.AI] https://arxiv.org/abs/2510.04542

[19] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. arXiv:2304.03442 [cs.HC] https://arxiv.org/abs/2304.03442

[20] Annu Rana and Gaurav Kumar. 2025. Model-First Reasoning LLM Agents: Reducing Hallucinations through Explicit Problem Modeling. arXiv:2512.14474 [cs.AI] https://arxiv.org/abs/2512.14474

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[22] John Schultz, Jakub Adamek, Matej Jusup, Marc Lanctot, Michael Kaisers, Sarah Perrin, Daniel Hennes, Jeremy Shar, Cannada Lewis, Anian Ruoss, Tom Zahavy, Petar Veličković, Laurel Prince, Satinder Singh, Eric Malmi, and Nenad Tomašev. 2025. Mastering Board Games by External and Internal Planning with Language Models. arXiv:2412.12119 [cs.AI] https://arxiv.org/abs/2412.12119

[23] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300* (2024).

[24] Yoav Shoham and Kevin Leyton-Brown. 2008. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, USA.

[25] Ardalan Tajbakhsh, Lorenz T. Biegler, and Aaron M. Johnson. 2024. Conflict-Based Model Predictive Control for Scalable Multi-Robot Motion Planning. arXiv:2303.01619 [cs.RO] https://arxiv.org/abs/2303.01619

[26] Hao Tang, Keya Hu, Jin Peng Zhou, Sicheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. 2024. Code Repair with LLMs gives an Exploration-Exploitation Tradeoff. arXiv:2405.17503 [cs.SE] https://arxiv.org/abs/2405.17503

[27] Qwen Team. 2024. Qwen2.5: A Party of Foundation Models. https://qwenlm.github.io/blog/qwen2.5/

[28] Emanuel Todorov, Tom Erez, and Yuval Tassa. 2012. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

[29] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. *arXiv preprint arXiv:2407.17032* (2024).

[30] Vivek Verma, David Huang, William Chen, Dan Klein, and Nicholas Tomlin. 2025. Measuring General Intelligence with Generated Games. arXiv:2505.07215 [cs.AI] https://arxiv.org/abs/2505.07215

[31] Di Wang, Zhenhua Wu, Yu Liu, Kai Chang, and Shaohua Wu. 2026. A Generalizable Framework for Building Executable Domain-Specific LLMs under Data Scarcity: Demonstration on Semiconductor TCAD Simulation. arXiv:2601.10128 [cs.CE] https://arxiv.org/abs/2601.10128

[32] Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. 2025. Co-Evolving LLM Coder and Unit Tester via Reinforcement Learning. arXiv:2506.03136 [cs.CL] https://arxiv.org/abs/2506.03136

[33] Lai Wei, Yuting Li, Kaipeng Zheng, Chen Wang, Yue Wang, Linghe Kong, Lichao Sun, and Weiran Huang. 2025. Advancing Multimodal Reasoning via Reinforcement Learning with Cold Start. arXiv:2505.22334 [cs.CL] https://arxiv.org/abs/2505.22334

[34] Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. 2024. Exploring Large Language Models for Communication Games: An Empirical Study on Werewolf. arXiv:2309.04658 [cs.CL] https://arxiv.org/abs/2309.04658

[35] Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does Reinforcement Learning Really Incentivize Reasoning Capacity in LLMs Beyond the Base Model? *arXiv preprint arXiv:2504.13837* (2025).

[36] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: Less Is More for Alignment. arXiv:2305.11206 [cs.CL] https://arxiv.org/abs/2305.11206

[37] Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, and Maarten Sap. 2024. SOTOPIA: Interactive Evaluation for Social Intelligence in Language Agents. arXiv:2310.11667 [cs.AI] https://arxiv.org/abs/2310.11667

Appendix A PSEUDOCODE

A.1 Perfect Information Game Generation Prompt

```
You are an expert Python programmer who is building the game of
      {game_name}.
Here is a description of the game:
{game_desc}

The goal is to implement Python functions with the following
      signatures and use the type definitions.
Your implementation MUST use module-level functions, NOT classes.
#START FUNCTION SIGNATURE
# Type definitions
Action = str
State = dict[str, Any]
PlayerObservation = dict[str, Any]

# Required Functions
def get_initial_state() -> State:
    """Returns the initial game state before any actions are
      taken."""
def apply_action(state: State, action: Action) -> State:
    """
    Returns the new state after an action has been taken.
    Ensure that the previous state is not mutated; always return a
      new state object.
    """
def get_current_player(state: State) -> int:
    """Returns current player (e.g. 0 or 1), or -4 for terminal
      state."""
def get_player_name(player_id: int) -> str:
    """Returns the name of the player."""
def get_rewards(state: State) -> list[float]:
    """Returns the rewards per player. May return non-zero values
      at non-terminal states if the game tracks running rewards
      (e.g., current scores or chip stacks); otherwise returns [0.0,
      0.0] until meaningful reward information is available."""
def get_legal_actions(state: State) -> list[Action]:
    """Returns legal actions for current state. Empty list if
      terminal."""
def get_observations(state: State) -> list[PlayerObservation]:
    """Returns [player_0_obs, player_1_obs]. For perfect info
      games, both see the same state."""
#END FUNCTION SIGNATURE

Use the type definitions.
All actions returned by get_legal_actions and accepted by
      apply_action must use ONLY the action strings specified in the
      game description above. Do not invent new action formats.
Ensure to import all libraries used.
Do not leave placeholders.
Do not repeat the function signature.
Do write comments explaining what the code does.
Do use helper functions to reduce code duplication.
Do NOT wrap the functions in a class. Define them at the module
      level.
```

A.2 Imperfect Information Game Generation Prompt

```
You are an expert Python programmer who is building the game of
      {game_name}.
Here is a description of the game:
{game_desc}

The goal is to implement Python functions with the following
      signatures and use the type definitions.
Your implementation MUST use module-level functions, NOT classes.
#START FUNCTION SIGNATURE
# Type definitions
Action = str
State = dict[str, Any]
PlayerObservation = dict[str, Any]

# Required Functions
def get_initial_state() -> State:
    """Returns the initial game state before any actions are
      taken."""
def apply_action(state: State, action: Action) -> State:
    """
    Returns the new state after an action has been taken.
    Ensure that the previous state is not mutated; always return a
      new state object.
    """
def get_current_player(state: State) -> int:
    """Returns current player (e.g. 0 or 1), or -4 for terminal
      state."""
def get_player_name(player_id: int) -> str:
    """Returns the name of the player."""
def get_rewards(state: State) -> list[float]:
    """Returns the rewards per player. May return non-zero values
      at non-terminal states if the game tracks running rewards
      (e.g., current scores or chip stacks); otherwise returns [0.0,
      0.0] until meaningful reward information is available."""
def get_legal_actions(state: State) -> list[Action]:
    """Returns legal actions for current state. Empty list if
      terminal."""
def get_observations(state: State) -> list[PlayerObservation]:
    """Returns [player_0_obs, player_1_obs]."""
def resample_history(obs_action_history:
      list[tuple[PlayerObservation, Action | None]], player_id: int)
      -> list[Action]:
    """
    Stochastically sample a valid sequence of actions (including
      'chance' outcomes) that explains the current observations.
    CRITICAL: The returned list must be a complete trajectory that
      can be replayed starting EXACTLY from get_initial_state().
    """
#END FUNCTION SIGNATURE

Use the type definitions.
All actions returned by get_legal_actions and accepted by
      apply_action must use ONLY the action strings specified in the
      game description above. Do not invent new action formats.
Ensure to import all libraries used.
Do not leave placeholders.
Do not repeat the function signature.
Do write comments explaining what the code does.
Do use helper functions to reduce code duplication.
Do NOT wrap the functions in a class. Define them at the module
      level.
```

A.3 Tiered Evaluation

```
# Tier weights (renormalized if information tier is skipped)
TIER_WEIGHTS = {"static": 0.15, "dynamics": 0.25, "information":
      0.30, "scenarios": 0.30}
def evaluate(code, game) -> float:
    module = load(code)

    # N is the number of games to fuzz on
    tiers = [static(module), dynamics(module, N=100),
      scenarios(module, game)]
    if game in IMPERFECT_INFO_GAMES:
        tiers.append(information(module, N=100))

    return mean(tier.score for tier in tiers)

def static(module) -> TierResult:
    yield syntax_ok # code compiled without error

    yield all(hasattr(module, f) for f in REQUIRED_API) # if the 6
      functions present

    if api_incomplete: yield [False] * 5; return

    state = module.get_initial_state()
    yield isinstance(state, dict)
```

```
    if state is None: yield [False] * 4; return
    yield returns list[str]:  module.get_legal_actions(state)

    yield returns list[float]: module.get_rewards(state)
    yield returns list: module.get_observations(state)
    yield returns int: module.get_current_player(state)

def dynamics(module, N) -> TierResult:
    # 4 binary checks - pass iff property holds across all N
      trajectories
    for _ in range(N):
        track no_crash # no exception
        track immutable # deepcopy(s) == s after apply_action(s,a)
        track deterministic
        track terminal_empty

def information(module, N) -> TierResult:
    if not hasattr(module, "resample_history"): yield [False] * 4;
      return

    for _ in range(N): # pid is player id (0 or 1)
        history = [(obs[pid], action) for each of pid's turns]

        resampled = module.resample_history(history, pid)

        replay resampled from get_initial_state(), at each pid turn:
            yield resample_legal: action in get_legal_actions(state)

            yield obs_reconstruction: get_observations(state)[pid]
      == recorded_obs

            yield action_consistency: resampled_action ==
      recorded_action

            yield resample_complete: all history entries consumed
      exactly

def scenarios(module, game) -> TierResult:
    for s in load_scenarios(game):
        state = replay(s.actions)
        yield terminal flag, current_player, rewards_sign, winner
      all match s.checks
```

A.4 GRPO Reward

```
def compute_reward(code, game, N=20, timeout=60) -> float:
    with sigalrm(timeout): return _reward(code, game, N)  # 0.0 on
      timeout (infinite loop)

def _reward(code, game, N) -> float:
    W = renormalized_weights(game) # no information tests for
      perfect info games
    module, err = load(code)

    if err: return 0.0

    # Tier 1
    s_static, can_continue = check_static(module)

    if not can_continue:
        return s_static * W["static"]

    # Tier 2 - Dynamics
    s_dyn = check_dynamics(module, N)

    if s_dyn < 0.5:
        return s_static * W["static"] + s_dyn * W["dynamics"]

    total = s_static * W["static"] + s_dyn * W["dynamics"]

    # Tier 3 - Scenarios  (fraction of golden scenarios passed)

    total += check_scenarios(module, game) * W["scenarios"]

    # Tier 4 - Information

    if "information" in W:
        if is_stub(module.resample_history): return total

        total += check_information(module, N) * W["information"]

    return total
```

Appendix B SCENARIO SAMPLES

B.1 Leduc Poker

Table 5: Evaluation scenarios for Leduc Poker.

Scenario	Action sequence	Outcome
P0 folds preflop	`deal:K, deal:Q, Fold`	P1 wins
P1 folds after P0 raises	`deal:K, deal:Q, Raise, Fold`	P0 wins
Showdown: K beats J	`deal:K, deal:J, Call, Call, deal:Q, Call, Call`	P0 wins
Pair beats non-pair (P0: K + public K)	`deal:K, deal:J, Call, Call, deal:K, Call, Call`	P0 wins
P0 folds postflop facing P1 raise	`deal:K, deal:Q, Call, Call, deal:J, Call, Raise, Fold`	P1 wins
Non-terminal after preflop deal	`deal:K, deal:Q`	P0 to act

B.2 Generalized Tic-Tac-Toe

Table 6: Evaluation scenarios for Generalized Tic-Tac-Toe.

Scenario	Action sequence	Outcome
P0 wins: top row, cols 0–3	`0,0` → `1,0` → `0,1` → `1,1` → `0,2` → `1,2` → `0,3`	P0 wins
P0 wins: column 0, rows 0–3	`0,0` → `0,1` → `1,0` → `0,2` → `2,0` → `0,3` → `3,0`	P0 wins
P0 wins: main diagonal	`0,0` → `0,1` → `1,1` → `0,2` → `2,2` → `0,3` → `3,3`	P0 wins
P1 wins: second row, cols 0–3	`0,0` → `1,0` → `2,0` → `1,1` → `3,0` → `1,2` → `4,0` → `1,3`	P1 wins
P1 wins: column 5, rows 2–5	`0,0` → `2,5` → `0,1` → `3,5` → `0,2` → `4,5` → `1,0` → `5,5`	P1 wins
Three-in-a-row is not terminal	`0,0` → `1,0` → `0,1` → `1,1` → `0,2`	P1 to act
Non-terminal after first move	`3,3`	P1 to act

Appendix C TRAINING HYPERPARAMETERS

Table 7: Training hyperparameters for SFT and GRPO. If not indicated, defaults were used.

Hyperparameter	SFT	GRPO	SFT+GRPO
General			
Base Model	Qwen2.5-3B-Instruct	Qwen2.5-3B-Instruct	Post-SFT Model
Precision	bfloat16	bfloat16	bfloat16
Epochs	2	2	2
Learning Rate	1×10^{-5}	2×10^{-5}	2×10^{-5}
LR Scheduler	Cosine	–	–
Warmup Ratio	0.03	–	–
Batch Size	1	1	1
Gradient Accumulation	4	4	4
Effective Batch Size	4	4	4
Optimizer	AdamW	AdamW 8-bit	AdamW 8-bit
Max Prompt Length	4096	4096	4096
Max Completion Length	4096	4096	4096
Gradient Checkpointing	–	✓	✓
LoRA Configuration			
LoRA Rank (r)	16	16	16
LoRA Alpha (α)	32	64	64
LoRA Dropout	0.05	0.05	0.05
Target Modules	`q,k,v,o_proj`	`q,k,v,o,gate,up,down_proj`	`q,k,v,o,gate,up,down_proj`
GRPO-Specific			
Num Generations (G)	–	8	8
Generation Batch Size	–	8	8
KL Penalty (β)	–	0.1	0.1
Max Gradient Norm	–	0.1	0.1
Temperature	–	0.6	0.6

Appendix D DATASET INFORMATION

Table 8: Games used in our dataset, categorized by train/eval usage, if it's Existing/OOD, and information type

Game	Existing/OOD	Information Type
Training Games		
Tic-Tac-Toe	Existing	Perfect
Connect4	Existing	Perfect
Pig	Existing	Perfect
Dots and Boxes	Existing	Perfect
Go	Existing	Perfect
Pentago	Existing	Perfect
Breakthrough	Existing	Perfect
Matching Pennies	Existing	Imperfect
Prisoner's Dilemma	Existing	Imperfect
Blackjack	Existing	Imperfect
Kuhn Poker	Existing	Imperfect
First Price Auction	Existing	Imperfect
Checkers	Existing	Perfect
Chess	Existing	Perfect
Liar's Dice	Existing	Imperfect
Goofspiel	Existing	Imperfect
Battleship	Existing	Imperfect
Hanabi	Existing	Imperfect
Leduc Poker	Existing	Imperfect
Backgammon	Existing	Perfect
Havannah	Existing	Perfect
Negotiation	Existing	Imperfect
Poker Hold'em	Existing	Imperfect
Evaluation Games (Held Out)		
Y	Existing	Perfect
Generalized Tic-Tac-Toe	OOD	Perfect
Generalized Chess	OOD	Perfect
Converge	OOD	Perfect
Gin Rummy	Existing	Imperfect
Hand of War	OOD	Imperfect
Quadranto	OOD	Imperfect

Appendix E ABLATION STUDIES

E.1 Ablation on Scenarios in Reward Function for GRPO

Table 9 shows the results of ablating scenarios from the reward function in GRPO. We opted to report results including scenarios in the main text due to higher overall results on each category.

Appendix F SAMPLES OF GENERATED CODE

F.1 SFT Qwen (Gin Rummy)

```
import copy
import random
from copy import deepcopy
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter

from typing import Any, List, Dict, Tuple
```

Table 9: Aggregate pass rates (%) across the four verification tiers (Static, Fuzzing, Semantics, Information) for only SFT+RLVR with and without scenarios. Values represent average pass rates over 5 samples at temperature 0.3.

		Test Hierarchies			
Game	**Model**	**Static**	**Fuzz**	**Semantics**	**Information**
Perfect Information Games					
Gen. Tic-Tac-Toe (OOD)	SFT+RLVR (w/o scenarios)	82.9%	75.0%	14.3%	N/A
	SFT+RLVR	100.0%	75.0%	22.9%	N/A
Converge (OOD)	SFT+RLVR (w/o scenarios)	88.6%	80.0%	0.0%	N/A
	SFT+RLVR	88.6%	75.0%	8.0%	N/A
Gen. Chess (OOD)	SFT+RLVR (w/o scenarios)	80.0%	75.0%	8.0%	N/A
	SFT+RLVR	80.0%	90.0%	12.0%	N/A
Y	SFT+RLVR (w/o scenarios)	97.1%	80.0%	24.0%	N/A
	SFT+RLVR	97.1%	80.0%	32.0%	N/A
Imperfect Information Games					
Quadranto (OOD)	SFT+RLVR (w/o scenarios)	94.3%	70.0%	16.0%	0.0%
	SFT+RLVR	85.7%	85.0%	12.0%	10.0%
Hand of War (OOD)	SFT+RLVR (w/o scenarios)	80.0%	60.0%	12.0%	0.0%
	SFT+RLVR	77.1%	75.0%	4.0%	0.0%
Gin Rummy	SFT+RLVR (w/o scenarios)	68.6%	75.0%	0.0%	0.0%
	SFT+RLVR	82.9%	75.0%	0.0%	0.0%
Aggregate Results					
	SFT+RLVR (w/o scenarios)	78.0%	75.7%	3.3%	0.0%
	SFT+RLVR	87.3%	79.3%	13.0%	3.3%

```
Action = str
State = Dict[str, Any]
PlayerObservation = Dict[str, Any]

def initialize_deck() -> List[int]:
    """Initialize a shuffled deck of cards."""
    suits = ['S', 'C', 'D', 'H']
    ranks = [str(i) for i in range(2, 11)] + ['J', 'Q', 'K', 'A']
    deck = [(rank, suit) for suit in suits for rank in ranks]
    import random
    random.shuffle(deck)
    return deck

def split_deck(deck: List[Tuple[str, str]]) ->
      Tuple[List[Tuple[str, str]], List[Tuple[str, str]]]:
    """Split the deck into stock and upcard."""
    return deck[:-2], deck[-2:]

def get_initial_state() -> State:
    """Returns the initial game state before any actions are
      taken."""
    deck = initialize_deck()
    stock, upcard = split_deck(deck)
    return {
        'deck': deck,
        'stock': stock,
        'upcard': upcard,
        'current_player': 0,
        'knocked': False,
        'knock_card': 10,
        'deadwood': {'player_0': 0, 'player_1': 0},
        'melds': {'player_0': [], 'player_1': []}
    }

def apply_action(state: State, action: Action) -> State:
    """Returns the new state after an action has been taken."""
    new_state = state.copy()
    if action == 'Draw stock':
        new_state['stock'], new_state['upcard'] =
      split_deck(new_state['deck'])
        new_state['current_player'] = (new_state['current_player']
      + 1) % 2
    elif action == 'Draw upcard':
        new_state['upcard'] = new_state['stock'].pop()
        new_state['current_player'] = (new_state['current_player']
      + 1) % 2
    elif action.startswith('Action: '):
```

```
        card = action.split(': ')[1]
        new_state['deck'].remove((card, new_state['upcard'][0]))
        new_state['stock'].append((card, new_state['upcard'][0]))
        new_state['current_player'] = (new_state['current_player']
      + 1) % 2
    elif action == 'Knock':
        new_state['knocked'] = True
        new_state['knock_card'] = 10
        new_state['knock_phase'] = 'Declare Melds'
    elif action == 'Done':
        new_state['knock_phase'] = 'Lay Off'
    elif action == 'Pass':
        new_state['knock_phase'] = 'Pass'
    return new_state

def get_current_player(state: State) -> int:
    """Returns current player (e.g. 0 or 1), or -4 for terminal
      state."""
    return state['current_player']

def get_player_name(player_id: int) -> str:
    """Returns the name of the player."""
    return f'Player {player_id}'

def get_rewards(state: State) -> List[float]:
    """Returns the rewards per player. May return non-zero values
      at non-terminal states if the game tracks running rewards
      (e.g., current scores or chip stacks); otherwise returns [0.0,
      0.0] until meaningful reward information is available."""
    return [0.0, 0.0]

def get_legal_actions(state: State) -> List[Action]:
    """Returns legal actions for current state. Empty list if
      terminal."""
    current_player = get_current_player(state)
    if state['knocked']:
        if state['knock_phase'] == 'Declare Melds':
            return ['Action: ' + card for card in state['deck']]
        elif state['knock_phase'] == 'Lay Off':
            return []
        else:  # 'Pass' or 'Done'
            return []
    else:
        if state['current_player'] == current_player:
            if state['knock_phase'] == 'Draw':
                return ['Draw stock', 'Draw upcard']
            else:
                return ['Draw stock', 'Draw upcard', 'Action: ' +
      card for card in state['deck']]
        else:
            return []

def get_observations(state: State) -> List[PlayerObservation]:
    """Returns [player_0_obs, player_1_obs]."""
    player_0_obs = {
        'deck': state['deck'],
        'stock': state['stock'],
        'upcard': state['upcard'],
        'deadwood': state['deadwood']['player_0'],
        'melds': state['melds']['player_0'],
        'knock_phase': state['knock_phase']
    }
    player_1_obs = {
        'deck': state['deck'],
        'stock': state['stock'],
        'upcard': state['upcard'],
        'deadwood': state['deadwood']['player_1'],
        'melds': state['melds']['player_1'],
        'knock_phase': state['knock_phase']
    }
    return [player_0_obs, player_1_obs]

def resample_history(obs_action_history:
      List[Tuple[PlayerObservation, Action | None]], player_id: int)
      -> List[Action]:
    """Stochastically sample a valid sequence of actions (including
      'chance' outcomes) that explains the current observations.
    CRITICAL: The returned list must be a complete trajectory that
      can be replayed starting EXACTLY from get_initial_state().
    """
    # Placeholder for resampling logic
    # For simplicity, we just return the most recent action
    return [action for _, action in obs_action_history]
```

F.2 SFT+RLVR Qwen (Quadranto)

```
import copy
import random
from copy import deepcopy
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter

import random

# Type definitions
Action = str
State = dict[str, Any]
PlayerObservation = dict[str, Any]

# Helper function to generate a random initial state
def get_random_initial_location():
    locations = [(0, 0), (0, 1), (1, 0), (1, 1), (0, 2), (0, 3),
      (1, 2), (1, 3), (2, 0), (2, 1), (3, 0), (3, 1), (2, 2), (2,
      3), (3, 2), (3, 3)]
    return random.choice(locations)

# Required Functions
def get_initial_state() -> State:
    """Returns the initial game state before any actions are
      taken."""
    # Initial state: Player 0 in Q1, Player 1 in Q4
    p0_loc = get_random_initial_location()
    p1_loc = get_random_initial_location()
    while p0_loc == p1_loc:
        p1_loc = get_random_initial_location()

    # Mapping of quadrant to coordinates
    quadrant_mapping = {
        "Q1": [(0, 0), (0, 1), (1, 0), (1, 1)],
        "Q2": [(0, 2), (0, 3), (1, 2), (1, 3)],
        "Q3": [(2, 0), (2, 1), (3, 0), (3, 1)],
        "Q4": [(2, 2), (2, 3), (3, 2), (3, 3)]
    }

    # Determine initial quadrants
    q0 = next((q for q, coords in quadrant_mapping.items() if
      p0_loc in coords), None)
    q1 = next((q for q, coords in quadrant_mapping.items() if
      p1_loc in coords), None)

    return {
        "p0_loc": p0_loc,
        "p1_loc": p1_loc,
        "current_player": 0,  # Player 0 starts first
        "q0": q0,
        "q1": q1
    }

def apply_action(state: State, action: Action) -> State:
    """
    Returns the new state after an action has been taken.
    Ensure that the previous state is not mutated; always return a
      new state object.
    """
    new_state = state.copy()

    if action == "place_p0:<row>,<col>":
        row, col = map(int, action.split(":")[1].split(","))
        new_state["p0_loc"] = (row, col)
        new_state["q0"] = next(q for q, coords in
      quadrant_mapping.items() if (row, col) in coords)
    elif action == "place_p1:<row>,<col>":
        row, col = map(int, action.split(":")[1].split(","))
```

```
        new_state["p1_loc"] = (row, col)
        new_state["q1"] = next(q for q, coords in
      quadrant_mapping.items() if (row, col) in coords)
    else:
        # Movement actions
        row, col = new_state["p0_loc"]
        if action == "Up":
            new_state["p0_loc"] = (row - 1, col)
        elif action == "Down":
            new_state["p0_loc"] = (row + 1, col)
        elif action == "Left":
            new_state["p0_loc"] = (row, col - 1)
        elif action == "Right":
            new_state["p0_loc"] = (row, col + 1)
        elif action == "Stay":
            pass

    # Check if the game ended
    if abs(new_state["p0_loc"][0] - new_state["p1_loc"][0]) <= 1
      and abs(new_state["p0_loc"][1] - new_state["p1_loc"][1]) <= 1:
        new_state["game_over"] = True
        new_state["winner"] = 1 if new_state["p0_loc"] ==
      new_state["p1_loc"] else 0
        new_state["reward"] = [new_state["winner"],
      -new_state["winner"]]
    else:
        new_state["game_over"] = False
        new_state["reward"] = [0.0, 0.0]

    return new_state

def get_current_player(state: State) -> int:
    """Returns current player (e.g. 0 or 1), or -4 for terminal
      state."""
    return state.get("current_player", -4)

def get_player_name(player_id: int) -> str:
    """Returns the name of the player."""
    return f"Player {player_id}"

def get_rewards(state: State) -> list[float]:
    """Returns the rewards per player. May return non-zero values
      at non-terminal states if the game tracks running rewards
      (e.g., current scores or chip stacks); otherwise returns [0.0,
      0.0] until meaningful reward information is available."""
    return state["reward"]

def get_legal_actions(state: State) -> list[Action]:
    """Returns legal actions for current state. Empty list if
      terminal."""
    if state["game_over"]:
        return []
    current_player = state["current_player"]
    legal_actions = []

    if current_player == 0:
        legal_actions.append("place_p0:<row>,<col>")
        legal_actions.append("Up")
        legal_actions.append("Down")
        legal_actions.append("Left")
        legal_actions.append("Right")
        legal_actions.append("Stay")
    else:
        legal_actions.append("place_p1:<row>,<col>")
        legal_actions.append("Up")
        legal_actions.append("Down")
        legal_actions.append("Left")
        legal_actions.append("Right")
        legal_actions.append("Stay")

    return legal_actions

def get_observations(state: State) -> list[PlayerObservation]:
    """Returns [player_0_obs, player_1_obs]."""
    p0_loc = state["p0_loc"]
    p1_loc = state["p1_loc"]
    p0_quadrant = state["q0"]
    p1_quadrant = state["q1"]

    p0_obs = {
        "My Loc": p0_loc,
        "Opponent Quadrant": p1_quadrant
    }
    p1_obs = {
        "My Loc": p1_loc,
        "Opponent Quadrant": p0_quadrant
    }

    return [p0_obs, p1_obs]

def resample_history(obs_action_history:
      list[tuple[PlayerObservation, Action | None]], player_id: int)
      -> list[Action]:
    """
    Stochastically sample a valid sequence of actions (including
      'chance' outcomes) that explains the current observations.
    CRITICAL: The returned list must be a complete trajectory that
      can be replayed starting EXACTLY from get_initial_state().
    """
    # For simplicity, we'll just return a deterministic sequence of
      actions based on the current state
    if player_id == 0:
        return ["place_p0:0,0", "Up", "Down", "Left", "Right",
      "Stay"]
    else:
        return ["place_p1:3,3", "Up", "Down", "Left", "Right",
      "Stay"]
```

F.3 RLVR Qwen (Quadranto)

```
import copy
import random
from copy import deepcopy
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter

from typing import *
import random

# Type definitions
Action = str
State = dict[str, Any]
PlayerObservation = dict[str, Any]

# Helper function to generate a random position within a quadrant
def random_position_in_quadrant(quadrant: str) -> tuple[int, int]:
    rows, cols = (0, 1), (0, 3), (2, 1), (2, 3)
    row, col = rows[quadrant[0]], cols[quadrant[1]]
    return random.choice([row, row + 1]), random.choice([col, col +
      1])

# Required Functions
def get_initial_state() -> State:
    # Initial positions
    p0_pos = random_position_in_quadrant('Q1')
    p1_pos = random_position_in_quadrant('Q4')

    # Initial state
    return {
        'p0_pos': p0_pos,
        'p1_pos': p1_pos,
        'turn_count': 0,
        'current_player': 0,
        'game_over': False
    }

def apply_action(state: State, action: Action) -> State:
    new_state = state.copy()
    p0_pos, p1_pos = new_state['p0_pos'], new_state['p1_pos']

    if action == 'place_p0:<row>,<col>':
        new_state['p0_pos'] = eval(action.split(':')[1])
    elif action == 'place_p1:<row>,<col>':
        new_state['p1_pos'] = eval(action.split(':')[1])
```

```
    else:
        if action == 'Up':
            new_state['p0_pos'] = (max(p0_pos[0] - 1, 0), p0_pos[1])
            new_state['p1_pos'] = (min(p1_pos[0] + 1, 3), p1_pos[1])
        elif action == 'Down':
            new_state['p0_pos'] = (min(p0_pos[0] + 1, 3), p0_pos[1])
            new_state['p1_pos'] = (max(p1_pos[0] - 1, 0), p1_pos[1])
        elif action == 'Left':
            new_state['p0_pos'] = (p0_pos[0], max(p0_pos[1] - 1, 0))
            new_state['p1_pos'] = (p1_pos[0], min(p1_pos[1] + 1, 3))
        elif action == 'Right':
            new_state['p0_pos'] = (p0_pos[0], min(p0_pos[1] + 1, 3))
            new_state['p1_pos'] = (p1_pos[0], max(p1_pos[1] - 1, 0))
        elif action == 'Stay':
            pass

    new_state['turn_count'] += 1
    new_state['current_player'] = 1 if new_state['current_player']
      == 0 else 0
    new_state['game_over'] = new_state['p0_pos'] ==
      new_state['p1_pos']

    return new_state

def get_current_player(state: State) -> int:
    return state['current_player']

def get_player_name(player_id: int) -> str:
    return f"Player {player_id}"

def get_rewards(state: State) -> list[float]:
    if state['game_over']:
        return [-1.0, 1.0]
    return [0.0, 0.0]

def get_legal_actions(state: State) -> list[Action]:
    current_player = state['current_player']
    legal_actions = []

    if current_player == 0:
        legal_actions.extend(['Up', 'Down', 'Left', 'Right',
      'Stay'])
    else:
        legal_actions.extend(['Up', 'Down', 'Left', 'Right',
      'Stay'])

    if state['turn_count'] < 20 and not state['game_over']:
        return legal_actions
    else:
        return []

def get_observations(state: State) -> list[PlayerObservation]:
    p0_pos, p1_pos = state['p0_pos'], state['p1_pos']
    p0_quadrant = 'Q1' if p0_pos in [(0, 0), (0, 1), (1, 0), (1,
      1)] else 'Q2' if p0_pos in [(0, 2), (0, 3), (1, 2), (1, 3)]
      else 'Q3' if p0_pos in [(2, 0), (2, 1), (3, 0), (3, 1)] else
      'Q4'
    p1_quadrant = 'Q1' if p1_pos in [(0, 0), (0, 1), (1, 0), (1,
      1)] else 'Q2' if p1_pos in [(0, 2), (0, 3), (1, 2), (1, 3)]
      else 'Q3' if p1_pos in [(2, 0), (2, 1), (3, 0), (3, 1)] else
      'Q4'

    return [
        {
            'loc': {'row': p0_pos[0], 'col': p0_pos[1]},
            'quadrant': p0_quadrant
        },
        {
            'loc': {'row': p1_pos[0], 'col': p1_pos[1]},
            'quadrant': p1_quadrant
        }
    ]

def resample_history(obs_action_history:
      list[tuple[PlayerObservation, Action | None]], player_id: int)
      -> list[Action]:
    # This function would need to be implemented based on the
      specific rules of the game and the history of
      observations/actions.
    # For simplicity, we'll just return a fixed sequence of actions
      that lead to a win for player_id.
    # In a real scenario, this function would need to be more
      complex and stochastic.
    if player_id == 0:
        return ['place_p0:0,0', 'place_p1:3,3', 'Up', 'Down',
      'Right', 'Up', 'Right']
    else:
        return ['place_p0:0,0', 'place_p1:3,3', 'Up', 'Down',
      'Left', 'Up', 'Left']
```

F.4 GPT-4o (Gin Rummy)

```
import copy
import random
from copy import deepcopy
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter

import random
from typing import Any, List, Tuple

# Type definitions
Action = str
State = dict[str, Any]
PlayerObservation = dict[str, Any]

# Constants
SUITS = ['S', 'C', 'D', 'H']
RANKS = ['A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J',
      'Q', 'K']
DECK = [rank + suit for suit in SUITS for rank in RANKS]
KNOCK_CARD_VALUE = 10
GIN_BONUS = 25
UNDERCUT_BONUS = 25

def get_initial_state() -> State:
    """Returns the initial game state before any actions are
      taken."""
    deck = DECK.copy()
    random.shuffle(deck)
    return {
        'deck': deck,
        'discard_pile': [],
        'hands': [deck[:10], deck[10:20]],
        'current_player': 0,
        'phase': 'Draw',
        'round_over': False,
        'scores': [0, 0]
    }

def apply_action(state: State, action: Action) -> State:
    """
    Returns the new state after an action has been taken.
    Ensure that the previous state is not mutated; always return a
      new state object.
    """
    new_state = state.copy()
    new_state['hands'] = [hand.copy() for hand in state['hands']]
    new_state['deck'] = state['deck'].copy()
    new_state['discard_pile'] = state['discard_pile'].copy()

    if action.startswith('Draw'):
        if action == 'Draw stock':
            card = new_state['deck'].pop(0)
        elif action == 'Draw upcard':
            card = new_state['discard_pile'].pop()
        new_state['hands'][new_state['current_player']].append(card)
        new_state['phase'] = 'Discard'
    elif action.startswith('Action:'):
        card = action.split(': ')[1]
        new_state['hands'][new_state['current_player']].remove(card)
        new_state['discard_pile'].append(card)
```

```
        new_state['current_player'] = 1 -
      new_state['current_player']
        new_state['phase'] = 'Draw'
    elif action == 'Action: Knock':
        new_state['round_over'] = True
    elif action == 'Pass':
        new_state['current_player'] = 1 -
      new_state['current_player']

    return new_state

def get_current_player(state: State) -> int:
    """Returns current player (e.g. 0 or 1), or -4 for terminal
      state."""
    if state['round_over']:
        return -4
    return state['current_player']

def get_player_name(player_id: int) -> str:
    """Returns the name of the player."""
    return f"Player {player_id}"

def get_rewards(state: State) -> List[float]:
    """Returns the rewards per player."""
    if not state['round_over']:
        return [0.0, 0.0]

    # Calculate deadwood and determine rewards
    deadwood_values = [calculate_deadwood(state['hands'][i]) for i
      in range(2)]
    if deadwood_values[0] < deadwood_values[1]:
        score = deadwood_values[1] - deadwood_values[0]
        state['scores'][0] += score
        return [score, 0.0]
    elif deadwood_values[1] < deadwood_values[0]:
        score = deadwood_values[0] - deadwood_values[1] +
      UNDERCUT_BONUS
        state['scores'][1] += score
        return [0.0, score]
    return [0.0, 0.0]

def get_legal_actions(state: State) -> List[Action]:
    """Returns legal actions for current state. Empty list if
      terminal."""
    if state['round_over']:
        return []

    if state['phase'] == 'Draw':
        actions = ['Draw stock']
        if state['discard_pile']:
            actions.append('Draw upcard')
        return actions

    if state['phase'] == 'Discard':
        hand = state['hands'][state['current_player']]
        actions = [f"Action: {card}" for card in hand]
        if calculate_deadwood(hand) <= KNOCK_CARD_VALUE:
            actions.append('Action: Knock')
        return actions

    return []

def get_observations(state: State) -> List[PlayerObservation]:
    """Returns [player_0_obs, player_1_obs]."""
    return [
        {
            'hand': state['hands'][0],
            'discard_pile': state['discard_pile'],
            'phase': state['phase']
        },
        {
            'hand': state['hands'][1],
            'discard_pile': state['discard_pile'],
            'phase': state['phase']
        }
    ]
```

```
def resample_history(obs_action_history:
      List[Tuple[PlayerObservation, Action | None]], player_id: int)
      -> List[Action]:
    """
    Stochastically sample a valid sequence of actions (including
      'chance' outcomes) that explains the current observations.
    """
    # This is a complex function that would require a detailed
      understanding of the game history.
    # For simplicity, we will return an empty list here.
    return []

def calculate_deadwood(hand: List[str]) -> int:
    """Calculates the deadwood value of a hand."""
    deadwood_value = 0
    for card in hand:
        rank = card[0]
        if rank in 'JQK':
            deadwood_value += 10
        elif rank == 'A':
            deadwood_value += 1
        else:
            deadwood_value += int(rank)
    return deadwood_value
```