

ML/RL Trading Bot Optimization Research

2.1 Gymnasium Environment Design

Summary

Designing a robust **trading environment** in Gymnasium requires careful attention to state representation, episode termination, and action abstraction. Financial environments often involve **variable-length episodes** due to market conditions or game rules, so the environment must handle episodes that can end at unpredictable times (e.g. on a rug pull). Best practices from trading simulators like FinRL and AnyTrading emphasize using realistic market dynamics and a rich observation space including price history and technical indicators ¹ ². A common approach for time-series data is to provide the agent with **temporal context**, either by *frame stacking* recent observations or using recurrent neural networks to maintain memory. Normalizing or scaling features is crucial – raw prices or balances can vary widely, so using percentage changes, log returns, or standardized values can stabilize training. Ensuring the observation space is **Markovian** (contains enough recent information to infer the state) often means including short-term history or indicators.

For **action space design**, trading tasks can use discrete actions (e.g. buy/sell/hold) or continuous actions (e.g. continuous position size). Discrete actions simplify learning but may limit precision, whereas continuous actions (or hybrid spaces) allow fine-grained control at the cost of more complex exploration ³. In this project, a MultiDiscrete space is used (action type, bet size, sell percent). This design is reasonable, but we should confirm it doesn't overly constrain the agent. Continuous position-sizing could be considered in the future (normalized to $[-1,1]$ as in FinRL's approach ³) if finer control is needed. **Episode management** can simply reset after each game (rug pull) – since all games eventually end, we treat each game as one episode. We must properly signal `done` on rug events and avoid artificial time limits that cut episodes short (unless needed to prevent excessively long runs) ⁴ ⁵. If episodes have a maximum length, use Gymnasium's `TimeLimit` wrapper to mark truncation distinctly from natural termination ⁴.

Recommendations

- **Observation Space:** Include a rich set of features capturing current market state, recent price trajectory, and derived indicators. Use either frame-stacking of last N price points or incorporate an LSTM/GRU policy to capture temporal dependencies. Normalize features (e.g. z-scores or min-max scaling) for stable learning.
- **Action Space:** Continue with a **discrete action space** for decision types (wait, buy, sell, etc.) and discrete sizing for simplicity. Ensure all meaningful actions are representable. In future, test a *continuous* action version for bet sizing if higher precision is needed, using SAC/TD3 algorithms that handle continuous actions.
- **Episode Termination:** End episodes on natural game end (rug pull or bankruptcy). Do not impose a fixed timestep limit unless needed; if you do, use Gym's truncation flags to differentiate timeouts

- ⁴ . Design the environment such that each episode corresponds to one game session, and handle variable lengths gracefully.
- **State Memory:** If not using a recurrent policy, consider stacking a short window of past observations (e.g. last 3–5 ticks) in the state to help the agent infer momentum and avoid partial observability. Alternatively, enable an **LSTM policy** in PPO (Stable Baselines3 supports recurrent PPO) so the agent can learn what to remember ⁶ .
- **Realism & Stochasticity:** Incorporate stochastic elements reflecting real game randomness (e.g. random rug timing drawn from empirical distribution). This prevents the agent from overfitting to specific recorded sequences and improves generalization.
- **Reference Designs:** Look at open-source trading environments (e.g. *Gym-AnyTrading*, *FinRL*) for inspiration on state and action design ¹ ³ . *FinRL*'s single-stock environment uses an action space of $\pm k$ shares (discrete) and a state of recent prices and technical indicators ³ , which aligns with our needs.

Tools/Libraries

- **Gymnasium** – Use Gymnasium's API for custom env, with wrappers like `VecNormalize` (from Stable Baselines) to normalize observations and rewards.
- **FinRL / OpenAI Baselines** – Reference implementations of trading environments (*FinRL*'s `StockEnvTrain`) ⁷ ⁸ for ideas on state/action design and reward calculation.
- **AnyTrading** – A Gym-based trading environment collection providing templates for forex, stocks, etc., which can be a baseline to compare with or test the agent in a simplified scenario.
- **Stable Baselines3 Recurrent Policies** – SB3 now offers LSTM policies (e.g. `RecurrentPPO`) that can maintain hidden state across timesteps, useful for variable-length sequences.
- **Data Normalization** – Use SB3's `VecNormalize` or custom normalization in the env (e.g., track running mean/std of features) to automatically standardize inputs.

Implementation Guidance

Integrate normalization and history into the environment. For example, wrap the environment with a `VecNormalize` to handle observation scaling automatically (this will compute running mean and std of each feature). If using frame stacking, you can use Gymnasium's `FrameStack` wrapper or modify the env's observation to concatenate recent states. Alternatively, switch the policy network to an LSTM: in SB3, this means using `MlpLstmPolicy` (or enabling the `use_recurrent=True` in the policy). Recurrent policies require resetting the LSTM state at episode boundaries – SB3 handles this if using their `RecurrentPPO`. Ensure the environment's `reset()` and `step()` are sending `done=True` at the correct rug pull event. For action space, the current `MultiDiscrete` approach can be left as is; just verify the mapping of indices to actual bet sizes or percentages is correct and possibly use a structured action (e.g., a named tuple) internally for clarity. It's important to test the environment in isolation (e.g., step through a dummy episode) to verify that state features (especially any involving history or positions) are computed correctly and that resetting clears old state. For variable episode length, you don't need special changes beyond proper `done` handling – PPO and other algorithms can handle episodes ending at different times, as long as the **Gymnasium** `done` and `truncated` flags are correctly set (use `truncated=True` only if using a max time limit wrapper) ⁴ . If you observe training instabilities, consider limiting extremely long episodes (e.g., if a game rarely lasts very long, you could cap at some percentile to avoid outliers – though in Rugs game, 100% eventually rug, so it's naturally bounded). Finally, maintain a consistent **observation schema** – since we have a complex Dict observation (89 features grouped into sections), ensure the agent sees a flat array or a structured input that the neural network can digest. If using SB3, a Dict observation will automatically

be flattened by policy (or you can use a custom policy to handle structured input). Test that each observation feature is finite and normalized (no unbounded values that could blow up network outputs).

References

- FinRL documentation on custom trading environments (single-stock example): emphasizes tailored state/action spaces and normalization ¹ ³ .
- Gymnasium documentation on **Time Limits** and episode truncation: proper handling of variable episode lengths ⁴ .
- Handmann *et al.* (2020) – comparison of feed-forward vs LSTM agent in trading; LSTM-based RL waited roughly **2× longer** before acting, indicating the value of memory for timing decisions ⁶ .
- **Gym Trading Env** (AnyTrading) – an open-source Gym environment for financial trading, can be consulted for design patterns ⁹ ¹⁰ .

2.2 Reward Shaping & Reward Hacking Prevention

Summary

Reward design in trading RL is critical and tricky. A well-shaped reward can accelerate learning, but any flaws can lead to *reward hacking*, where the agent exploits the reward function in unintended ways ¹¹ ¹² . In our case, the initial 17-component reward function was too complex and had loopholes (e.g. rewarding SELL even with no position), leading the agent to maximize reward without actually trading (e.g. spamming SELL for points). This is a classic outcome of mis-specified rewards, as the agent finds a clever “hack” to get reward without achieving the true goal ¹¹ . The simplified 2-component reward (pure P&L and bankruptcy penalty) is much closer to the true objective, reducing opportunities for exploitation. In financial RL, *dense vs sparse rewards* must be balanced: a purely profit-at-end reward (sparse) aligns perfectly with the goal but makes credit assignment difficult, whereas adding intermediate rewards (dense) can guide the agent but may introduce proxy objectives that can be gamed ¹² . A key principle from literature is to use **potential-based shaping** – additional reward terms that are shaped as differences of a potential function between states – which guarantees not to change the optimal policy ¹³ . This can inject guidance (e.g. encourage reaching a certain multiplier zone) without introducing new exploits, as long as the shaping is potential-based (thus theoretically ensuring no new loopholes in the optimal policy ¹³).

Preventing reward hacking systematically involves both **better reward design** and **monitoring**. Reward functions should be as aligned as possible with the true goal (profit) and any extra term must be carefully vetted. Techniques like **unit tests for rewards** (simulate scenarios to see if undesired behavior is rewarded) can catch bugs early. Additionally, one can incorporate *regularization* in the learning process (entropy bonuses, action penalties) to discourage extreme behaviors. Entropy bonus (already in PPO) helps keep the policy probabilistic and can prevent it from converging to a degenerate exploit too quickly. Another approach is using **intrinsic rewards or constraints** – for example, add a small penalty for each action to discourage frivolous actions, or a penalty for holding too long to avoid infinite hold exploits. Monitoring during training is essential: track action distribution and ensure a healthy mix (e.g. not 80% SELL with 0% BUY, which was a red flag). If the agent finds a new exploit (e.g. endlessly waiting to avoid risk), we may need to adjust rewards or environment rules accordingly. In research, *anomaly detection* methods have been proposed to detect reward hacking by comparing the agent’s behavior to a trusted reference policy ¹⁴ , but a simpler practical approach is to evaluate the agent on **true performance metrics** (like actual profit) frequently and ensure reward score correlates with them.

Recommendations

- **Keep Rewards Aligned:** Favor reward components that directly reflect the true objectives (e.g. realized profit, losses). Minimize or remove proxy rewards that don't strictly tie to performance (e.g. "survival time" or "zone entry" bonuses) unless they are formulated in a potential-based way that doesn't conflict with maximizing profit ¹² ¹³.
- **Potential-Based Shaping:** If adding shaping rewards, use potential-based reward shaping (PBRS) methods ¹³. For instance, define a potential function $\Phi(s)$ (e.g. based on distance to target multiplier or remaining risk) and shape reward as $\gamma \Phi(s') - \Phi(s)$. This provides dense feedback without changing optimal behavior.
- **Test for Hacks:** Create specific test scenarios (either via custom env or analytical reasoning) to see if the reward would incentivize unrealistic actions. For example, test that taking no positions yields zero reward (not positive), or that SELL yields no reward if no holdings. Ensure **no action has free reward** without corresponding risk.
- **Reward Scaling & Clipping:** Continue using reward clipping (e.g. ± 1000) to prevent outliers from destabilizing training. Clip only to prevent explosions, not so much that it hides important differences.
- **Dense vs Sparse Mix:** Consider a *hybrid reward*: primarily use **sparse reward = final profit**, but potentially add a small time-step penalty or cost of holding to encourage timely exits (careful to not reintroduce exploits). Alternatively, give a tiny positive reward for each successful exit before rug (which is essentially realized profit anyway). Keep any dense reward small relative to profit so it doesn't dominate.
- **Regularization:** Maintain an entropy bonus (the 0.01 is fine) to keep exploration. You can also add a mild penalty for each trade action (to discourage excessive churn) or each tick in a position (to discourage never selling). These act as regularizers to avoid extreme behaviors like endlessly holding or rapid-fire trading.
- **Monitor & Iterate:** Use training callbacks to log the percentage of each action taken and the agent's profit in evaluation episodes. If you notice a weird skew (like 0 buys), intervene by reviewing the reward. Be prepared to iterate on the reward function as the agent finds corner cases. It's often an iterative tuning process.
- **Imitation/IRL:** Consider **inverse RL or apprenticeship learning** to infer a reward function from expert demonstrations. If specifying a good reward manually is hard, algorithms like GAIL (Generative Adversarial IL) effectively *learn* a reward that replicates expert behavior. This learned reward might capture subtle preferences (like avoiding certain risks) that a hand-crafted reward misses. However, ensure the learned reward is interpretable and doesn't itself have loopholes.

Tools/Libraries

- **Reward Function Testing:** Use simple scripts or unit tests (e.g. in `pytest`) to feed the reward calculator specific state-action sequences. The connected `reward_calculator.py` can be tested by simulating a scenario (e.g. `positions=0, agent sells` should yield 0 reward).
- **TensorBoard/WandB:** Monitor custom metrics during training (like action counts, average holding time, etc.). Weights & Biases (wandb) is particularly useful to set up custom charts for these and can alert if the agent's behavior diverges from expectations.
- **RL Reward Design Libraries:** Though not common, libraries like *rllab* or *OpenAI SpinningUp* provide guidance on shaping. The `imitation` library (by SB3 team) can be used for inverse RL (GAIL, AIRL) to derive rewards from demonstrations if needed.

- **Anthropic’s RLHF tools:** (If applicable) Libraries from human-feedback RL could help detect odd reward exploitation behaviors by comparing policy actions to a reference (though likely overkill here).
- **Potential-based Shaping Reference:** The 1999 Ng et al. paper (potential-based shaping) is a guiding reference ¹³ – though not a library, implementing its formula ensures safe shaping.

Implementation Guidance

Implement the *minimal reward config* (financial + bankruptcy) as the default. Gradually introduce any new component one at a time and observe the effect. For example, if you want to re-introduce a *rug avoidance reward*, implement a check that only gives that bonus if the agent actually has an open position that it exits before a rug. Then test it with an episode where the agent holds a position and exits versus one where it never held a position – confirm only the former gets the reward. A practical technique is to run a **random or scripted agent** in the environment to see what reward it accumulates – this can reveal if doing nothing or doing something silly yields reward.

To systematically prevent reward hacking, you can incorporate checks in the training loop: after each training iteration (or every N episodes), evaluate the agent’s actual profit over, say, 20 simulation games. If the reward curve is high but profit is near zero or negative, that indicates reward misalignment. This can trigger an automatic review or early stopping. In the long term, one can train a *reward model* via human preference (RLHF style) to judge the agent’s performance on true objectives, but that’s complex.

Another idea is **“tripwire” states** ¹⁵ ¹⁶ – embed certain hidden conditions in the environment that should never be actively sought by a truthful agent. If the agent consistently triggers them for reward, you catch it. For instance, you might add a hidden flag that if an agent sells with zero position 10 times, it triggers a special log. This is more of a debugging trick.

From a theoretical perspective, rely on proven frameworks: *Conservative reward design* (only reward what you absolutely want) and *regular audits*. The emergent mind summary suggests multi-objective normalization and adversarial training to detect reward manipulation ¹⁷ ¹⁸ – in practice, just keep the reward function simple and directly tied to money made.

In summary, **test early, test often**: every time you tweak the reward, run a short training and see if the agent’s behavior aligns with intuition (does it actually trade and make money?). If it doesn’t, roll back or adjust, because the further a bad reward goes, the harder it is to fix the learned policy.

References

- Lilian Weng’s *“Reward Hacking in Reinforcement Learning”* – highlights how agents exploit poorly designed rewards and the importance of correct shaping ¹¹ ¹² . Introduces potential-based shaping from Ng et al. (1999) for safe reward adjustments ¹³ .
- *Concrete Problems in AI Safety* (Amodei et al. 2016) – defines reward hacking and suggests mitigations like careful reward design and monitoring for unintended behaviors ¹⁹ ²⁰ .
- Emergent Mind on *Avoiding Reward Hacking* – recommends constrained optimization and dynamic diagnostics to keep agents aligned ¹⁷ ¹⁸ . Emphasizes multi-objective approaches to reduce over-optimization of a single proxy.

- Srivastava *et al.* (2025) “Risk-Aware RL Reward for Trading” – proposes a multi-component reward (return and risk terms) and notes that single-metric rewards (like Sharpe or return alone) can encourage reward hacking or over-optimization ²¹ ²². Using multiple weighted terms can mitigate exploits by balancing objectives ²¹ (though complexity must be handled carefully, as in our case).
- Pan *et al.* (2022) – anomaly detection for reward hacking: suggests comparing agent policy to a trusted policy to flag misalignment ¹⁴ (conceptually interesting for future, though currently a simpler approach suffices).

2.3 Stable Baselines 3 & PPO Optimization

Summary

Proximal Policy Optimization (PPO) via Stable Baselines3 is our current learning algorithm. PPO is generally a solid choice for continuous or discrete action spaces, but its default hyperparameters may not be optimal for a trading scenario, which often has noisy, non-stationary reward signals. Financial environments can require longer horizon credit assignment and high stability. Key PPO hyperparameters to consider are the learning rate, batch size (number of timesteps per update), discount factor, and entropy coefficient. **Learning rate** $\sim 3e-4$ (our current) is a reasonable starting point; some trading applications find a smaller LR (e.g. $1e-4$ or $5e-5$) helps stabilize training due to noisy gradients. **Batch size/rollout length** (`n_steps` in PPO) should be large enough to capture an episode or significant portion – perhaps on the order of the average game length. If the median game is ~ 138 ticks, using `n_steps` around 128-256 could work (ensuring multiple updates per episode). A larger batch (e.g. 1024) might improve gradient estimates but also smooth out rare events. **Gamma (discount)** likely should be high (approaching 1) since we care about long-term outcome (final profit) – using $\gamma=0.99$ or even 0.995 to value rewards up to hundreds of ticks in the future. Too low gamma would make the agent myopic (not good for trading where best rewards come later). **Entropy coefficient** 0.01 is fine to encourage exploration; it can be tuned down gradually as training progresses (some schedules anneal entropy to zero to allow convergence).

In terms of algorithm choice, **off-policy methods** like SAC or TD3 might be worth exploring. PPO (on-policy) requires lots of fresh samples; in a limited data or slow environment, that can be a bottleneck. Off-policy algorithms reuse experience and can be more sample-efficient – for instance, prior work found TD3 converged faster than PPO in stock trading tasks ²³ ²⁴. SAC is an attractive option if we switch to continuous actions (it excels in continuous domains and can handle stochastic policy with exploration). For discrete actions, we could consider DQN or its variants, but our multi-discrete action (and need for continuous sizing) leans toward either *multi-discrete PPO* (current approach) or converting to continuous with SAC/TD3. Another algorithm to mention is **A2C/A3C** (which is like a simpler PPO without the clipping) – not as stable as PPO but sometimes works with fewer tuning parameters, or *DQN* with discretized actions if we handle multi-dimensional actions carefully. However, PPO’s combination of stability and parallelism is likely still a good choice to continue with, as long as we tune it.

To improve PPO’s performance, using **multiple parallel environments** is key. PPO in SB3 by default uses a single environment if not told otherwise. We should use `SubprocVecEnv` or `DummyVecEnv` with, say, 8 or more parallel game simulations. This feeds PPO with more diverse experiences per update and speeds up training wall-clock time. A typical setting is 8 to 16 envs for a reasonably fast environment. If each game is short (a couple hundred steps), we can definitely run e.g. 8 envs \times 256 steps per update = ~ 2048 steps per update (which is a common PPO batch size).

Training monitoring: We should implement callbacks for saving models periodically (checkpointing) and early stopping if needed. In RL, early stopping is tricky (because reward might be noisy), but we can watch a metric like average profit or success rate. If after X timesteps the profit hasn't improved in a long time, we might stop or adjust hyperparams. Checkpointing every certain number of timesteps (e.g. every 50k steps) to disk is recommended so we can roll back to the best model observed. SB3 provides a `EvalCallback` to evaluate on a test environment and save the best model by a chosen metric. We can use that to ensure we keep the best-performing policy (e.g. measured by average ROI in evaluation games).

Finally, consider **training duration:** For a complex task like this, PPO often needs on the order of millions of timesteps. Our roadmap mentioned 500k–1M timesteps; that is a good ballpark. We might start seeing reasonable behavior by a few hundred thousand steps, but going to 1M or more could yield further improvement as long as we're not overfitting to the environment. If the environment has some stochasticity, overfitting is less of a concern than in supervised learning, but we still should avoid training so long that the agent memorizes quirks of our simulator (especially if it's based on historical data).

Recommendations

- **Hyperparameter Tuning:** Use systematic tuning (e.g. grid search or Bayesian optimization via Optuna) for PPO hyperparams. Focus on learning rate (try 1e-3, 3e-4, 1e-4, etc.), batch size/n_steps (128, 256, 512), and gamma (0.99 vs 0.995). The **RL Baselines3 Zoo** provides tuned hyperparams for some tasks; leverage those as a starting point for trading.
- **Parallel Environments:** Run **8–16 parallel envs** for PPO. This improves sample throughput and stability (the gradient is averaged over more diverse samples). Use `SubprocVecEnv` if environment step is CPU-intensive, or `DummyVecEnv` if not (Dummy is fine if each step is fast, as it avoids inter-process communication overhead).
- **Alternate Algorithms:** Experiment with **SAC** or **TD3** if continuous action space is feasible (e.g. represent "action_type + magnitude" as continuous vector). SAC provides efficient learning with a replay buffer, which can make better use of limited data. If sticking to discrete actions, try **DQN** or **Rainbow** on a simplified action version (though multi-discrete is not directly supported by vanilla DQN). **Ray RLlib** could be used to handle multi-discrete with custom models or to try policy-gradient and Q-learning variants easily.
- **Callback & Checkpointing:** Use SB3's `EvalCallback` to evaluate the policy every N timesteps on a set of evaluation games (either a fixed set of recorded games or random seeds) and save the model when it achieves a new high score (e.g. best average profit or highest win rate). This ensures we keep the best model even if training later diverges. Implement a custom callback to log **action distribution, average holding time**, etc., to catch any emerging exploit early (ties into reward monitoring from 2.2).
- **Learning Rate Schedule:** Consider a **linear learning rate decay** – start at 3e-4 and linearly drop to 0 over the course of training (SB3 allows schedule functions). This often helps PPO converge to a stable policy at the end (high LR early for exploration, low LR later for fine-tuning). Alternatively, use cyclical or adaptive optimizers if needed, but linear annealing is simple and effective.
- **Early Stopping Criteria:** Define some criteria like "if average episode profit over last 100k steps hasn't improved by more than X, consider stopping or reducing LR." RL doesn't have a straightforward validation loss, but we can use our profit metrics for this. This prevents wasting time if the agent plateaus.
- **Total Timesteps:** Plan for at least on the order of **10⁶ timesteps** of training for a complex task. It's good to evaluate at intermediate points (e.g. 100k, 200k...) to see progress. Sometimes training

longer can cause performance to degrade (if agent forgets or overfits); that's another reason to use checkpoints. We should also maintain a **validation set** of games or a live simulation to periodically test the policy without training, ensuring it generalizes.

- **Explore Hindsight/Replay:** Although PPO is on-policy, we can augment it with a **replay buffer for off-policy fine-tuning** (SB3's PPO cannot directly use replay, but algorithms like ACER or using the demonstration data in a replay buffer via SQIL could help). If data is sparse, leveraging a replay of past experiences or mixing in demonstration experiences (as additional training data) can improve sample efficiency.

Tools/Libraries

- **Stable Baselines3 + RL Baselines3 Zoo:** The Zoo repository contains presets and Optuna scripts for tuning. We can use it to run hyperparameter searches specifically for our environment, which will automatically try variations of PPO/SAC parameters and report the best ²⁵ ²³.
- **Optuna or Ray Tune:** These libraries can be integrated to run parallel experiments for hyperparam tuning. Ray Tune, for example, can work with RLlib or even SB3 (via a wrapper) to sweep parameters and use schedulers for early stopping bad runs.
- **TensorBoard & SB3 Monitor:** Use SB3's built-in TensorBoard logging. Track custom metrics (we can log profit or Sharpe ratio as a custom scalar if we compute it in a callback). The SB3 `Monitor` wrapper on the env can record per-episode reward/profit which is then accessible for analysis.
- **Ray RLlib:** If we want to try an alternative to SB3, RLlib is a scalable RL library that supports multi-agent and large-scale training. It could run PPO, DQN, etc., and offers advanced features (e.g. imitation learning integration, multi-GPU support). For now, SB3 is simpler, but RLlib might be used in Phase 2 (live deployment) if we need more scalability.
- **CleanRL:** Another library with single-file implementations of PPO, SAC, etc., with sensible defaults and easy logging (including weights & biases integration by default). It's good for quick experiments to confirm if an algorithm change might help.
- **Checkpointing:** Use Python's `os.makedirs` and SB3's `model.save()` to save models. SB3 doesn't automatically keep many checkpoints, but you can handle that in a callback (e.g., save with timestamp or step count in filename). Also record hyperparams with each model (SB3's save includes them, but if using custom training loop, keep a JSON log).
- **Evaluation Environment:** Create a deterministic evaluation environment (maybe with a fixed random seed or recorded games) for consistent benchmarking. You might build a small wrapper around the env that, for example, plays through a set of 10 known game seeds and measures outcomes. This can be invoked periodically to measure generalization.

Implementation Guidance

In `rugs-rl-bot/scripts/train.py` (or wherever training is run), incorporate `make_vec_env` with `n_envs=8`. For example:

```
env = make_vec_env(RugsTradingEnv, n_envs=8, env_kwargs={...})
env = VecMonitor(env) # to track episode returns
env = VecNormalize(env, norm_obs=True, norm_reward=False) # normalize
observations
```

```
model = PPO("MlpPolicy", env, n_steps=256, batch_size=64, learning_rate=3e-4,
gamma=0.995, ent_coef=0.01, tensorboard_log="./logs/")
```

Adjust `n_steps` such that `n_steps * n_envs` is a multiple of `batch_size` (SB3 PPO requires that for proper minibatching). In the above, $256 * 8 = 2048$ total, which is divisible by 64, good. Monitor training via TensorBoard: SB3 will log reward by default. Use a callback for evaluation:

```
eval_env = make_vec_env(RugsTradingEnv, n_envs=1, env_kwargs={"record": True})
# maybe record a fixed sequence
eval_callback = EvalCallback(eval_env, n_eval_episodes=10, eval_freq=50000,
best_model_save_path='./logs/best_model', deterministic=True, render=False)
```

Attach `eval_callback` to `model.learn()`. This will test the model every 50k steps on 10 evaluation games and save it if it's the best so far. We can extend this callback to log custom things like Sharpe ratio (compute from rewards or profit series) or action counts.

For algorithm switching, if trying SAC, we'd convert action space to Box (maybe two continuous outputs: one for betting amount (0 to 0.5 SOL) and one for sell fraction (0-1). We could keep discrete buy/sell decision by treating it as a continuous value that we interpret (e.g. output $>0 \Rightarrow$ buy, $<0 \Rightarrow$ sell, magnitude for size). This would be a more complex refactor, so likely stick with PPO until basic profitability is achieved, then consider SAC/TD3 in later phases. The FinRL results ²³ ²⁴ showing TD3 performing well suggests that once we have a working PPO agent, it may be worth implementing TD3 for comparison – Stable Baselines3 has `TD3` class, so it's doable if we define a continuous action version of the env.

Make sure to use **proper seeding** for reproducibility when tuning – SB3 allows setting seeds on model and env. This will help ensure comparability between runs. Also, watch out for **overfitting**: if you train on a fixed set of recorded games, the agent might memorize them. Mitigate this by introducing randomness in simulations (which I believe we do via random rug events). If we have a **validation set of games** (some recorded sessions not used in training), we should definitely evaluate the agent on them periodically. If performance on training games keeps rising but on validation games plateaus or drops, that's a sign of overfitting. Then we might incorporate more randomness or limit training time.

References

- FinRL Single Stock example – compares PPO, A2C, DDPG, TD3 on stock trading; notes that **TD3 converges faster and can achieve higher reward** within 100k timesteps ²³ ²⁴ . Implies off-policy may be more sample-efficient for trading.
- Stable Baselines3 **"RL Tips and Tricks"** – documentation suggesting that default hyperparams might need tuning for new environments ²⁶ . Recommends careful adjustment of learning rate, etc., and using multiple envs for PPO.
- Stable Baselines3 EvalCallback – for model checkpointing and evaluation (official SB3 docs). This is key for tracking true performance.
- **Optuna for SB3** – Antonin Raffin's blogs or RL Zoo docs show how to integrate Optuna to tune hyperparameters, which can save significant time finding the best settings.
- Reddit discussion on PPO for Bitcoin with SB3 – highlights practical issues (like reward scaling) and suggests tuning parameters for financial data (user experiences).

2.4 Imitation Learning from Human Demonstrations

Summary

We have 204 recorded human **ButtonEvent demonstrations** (expert actions) with full context, which is a valuable but relatively small dataset. Imitation Learning (IL) can leverage this to jump-start training. **Behavioral Cloning (BC)** is the simplest IL: treat the problem as supervised learning, mapping states to the human actions. BC can quickly learn the demonstrators' behavior, but it often suffers from compounding errors – if the agent drifts off the states seen in the demos, it may not know how to recover (covariate shift). With only 204 samples, BC might not fully cover the state space, so the agent could latch onto some partial strategies but then get confused outside those scenarios. More advanced IL like **Generative Adversarial Imitation Learning (GAIL)** uses demonstrations to train a discriminator that guides the agent to mimic expert behavior. GAIL tends to require fewer expert trajectories than pure BC and accounts for the state distribution shift by integrating with RL (it effectively learns a reward that the expert would maximize) ²⁷ ²⁸ . However, GAIL is heavy to train (needs adversarial training and lots of environment interaction) ²⁹ ³⁰ . A promising approach is to **combine IL with RL**: use BC to initialize the policy (so it starts near the expert behavior), then fine-tune with RL (PPO) on the actual reward. This way, the agent doesn't start from scratch and hopefully inherits good habits (like entering around 25-50x and exiting before crash, if the demos reflect that). Another approach is *Concurrent IL+RL*: e.g. **Dagger (Dataset Aggregation)** which would require querying an expert during training whenever the agent deviates, to get corrections. Since we likely can't query a human in real-time, Dagger isn't directly applicable unless we simulate an expert (maybe the sidebet model or a rule-based strategy stands in as an "expert" to provide corrective actions).

Given the limited demo data, **data augmentation** can help. We might break each demonstration trajectory into multiple state-action pairs, or add slight noise to states to increase data (though one must be careful not to mislabel actions to significantly different states). Also, we should leverage the full context captured: perhaps each ButtonEvent has the entire game state – this could be used to engineer additional training points (e.g. if the human didn't press a button at some tick, that implies at those states the "do nothing" action was taken; we could treat those as implicit demonstrations of WAIT action). This effectively could generate many more state->action samples (every tick where the human did *not* act is a data point for the "no action" decision). This kind of **inverse labeling** needs caution but could greatly expand training data from 204 events to potentially thousands of state-action examples.

In terms of IL algorithms: **BC** is easy to implement with 204 samples (just train a policy network on those). **GAIL** could work since it's been shown to learn from small numbers of trajectories by interaction ²⁹ , but it will require training the agent in the environment with a GAN-like loss, which might be time-consuming. A compromise is **BC + RL fine-tuning (pretraining)**, which often yields faster convergence than RL from scratch. Another is **Reward shaping from demos**: use inverse RL (MaxEnt IRL) to infer a reward function that would make the demos optimal, then use that reward in RL. But that's quite complex to implement given time. Another method, **SQIL (Soft Q Imitation Learning)**, treats all demo transitions as having a fixed positive reward and non-demo transitions as a small negative reward, then runs RL (like DQN) to learn a policy that reproduces the expert behavior ³¹ . SQIL has been shown to perform comparably to GAIL and is simpler: it basically forces the agent to include the expert transitions in its experience replay with high reward ³¹ . This could be an interesting way to incorporate demos for an off-policy algorithm.

We should also consider that human demos might be suboptimal or noisy (maybe not all their actions were perfect). If the demonstrations have some mistakes or variance, pure imitation might copy those flaws.

Combining with RL can allow improvement beyond the expert. This is known as going from **apprenticeship to mastery** – e.g. using IL to get to expert level, then RL to potentially exceed it by exploiting dynamics humans couldn't. For example, maybe the human was a bit conservative; the RL fine-tune might find more aggressive profit-taking if it yields higher reward.

Recommendations

- **Behavioral Cloning Pretraining:** Perform supervised learning on the 204 demonstration points to initialize the policy. Even though it's small, it can teach basic notions (like *when to click sell*). Use data augmentation if possible (e.g. include states with no action as examples of "WAIT"). This gives a starting policy π_0 that can then be used in PPO training instead of random initialization.
- **Dagger-like Offline Augmentation:** Augment the demo dataset by rolling out partial trajectories: for each demonstration, you have the state where action was taken – also include a few states prior to that (with the action "WAIT") if the human was waiting, and states after (till next action or game end) labeled appropriately. Essentially reconstruct the sequence of human decisions in each game. This expanded sequence can then train a policy to mimic the full strategy, not just the singular button presses.
- **GAIL or Adversarial IL:** If time permits, use an IL library (like SB3's *imitation* library) to run GAIL. GAIL would learn a reward function that the expert trajectories get high reward and use PPO to train the agent to maximize that. This can incorporate the demo info more fundamentally. However, ensure the adversarial training has enough environment interaction budget (it can be sample-inefficient).
- **Combine IL and RL:** After IL pretraining, run PPO with a relatively low learning rate initially to avoid destroying the pretrained policy. You can also use a **loss function mixture** during early training: e.g., optimize a combined loss = RL loss + λ supervised loss on demo actions (for those states we have) ³². This is sometimes called *Behavior Cloning regularization* and can stabilize training (the agent stays near demonstration behavior while exploring). Gradually reduce λ as training goes on, letting the agent deviate if it finds better strategies.
- **Leverage Sidebet Model as Expert:** If the sidebet predictor is very good at timing exits (754% ROI suggests it's profitable), we could generate **synthetic demonstrations** using a rule: e.g. create an automated policy that buys in the sweet spot and exits when `should_exit` from sidebet triggers. Use that policy to simulate a bunch of games (in our environment) and record its actions as additional training data. This could massively increase the demonstration dataset with a presumably strong strategy. Even if not perfect, it gives the IL something to chew on beyond the 204 human actions.
- **Quality over Quantity:** Ensure the demonstrations are high-quality (successful trajectories). If some of the 204 demos were actually poor plays (maybe a user misclicked or got rugged early), consider filtering them out so the agent learns from the best. If that's hard to determine, at least weight each demo maybe by its outcome (successful rounds could be given higher weight in BC loss).
- **Architectures for IL:** We could consider using a **Transformer policy** that might ingest the whole trajectory, but given only 204 points, that's not feasible to train from scratch. Instead, an LSTM policy could be beneficial if demonstrations involve temporal dependencies (like "wait until tick X then sell"). An LSTM could in principle be trained via BC to reproduce sequences of actions given sequences of observations. If we have full trajectories of demonstrations (not sure if we do or just individual actions), training an LSTM on them might capture the timing aspect. For now, a standard feed-forward network is fine for BC with maybe some history features included.

- **Data Requirements:** Try to get more demos if possible. But if not, maximize the information from current ones (as described with augmentation). 204 data points is extremely low for training a neural policy, so **don't expect BC alone to yield a perfect policy**. Its main use is to provide a sensible initialization.

Tools/Libraries

- **SB3 *imitation* library:** Provides implementations of BC, GAIL, and even DAGGER. For instance, `imitation.algorithms.bc.BC` can train a policy on demonstration data easily. `imitation.algorithms.adversarial.gail` can run GAIL given expert trajectories and an environment ³³ ³⁴. This library will allow integrating with our SB3 PPO policy seamlessly (since it can use SB3 policies internally).
- **Dataset Management:** Use PyTorch or TensorFlow to do behavioral cloning. For example, extract the observation vectors for each demo state and the action taken, build a small dataset and train the model's policy network (you can directly use the same architecture as the PPO policy). The *imitation* library's BC or simply writing a quick supervised training loop with MSE or cross-entropy on actions is fine.
- **Expert Trajectory Format:** We might need to convert our ButtonEvent logs into a format the IL library expects (like a list of (obs, act) pairs or trajectory objects). The *imitation* lib has utilities to ingest expert data. Alternatively, we can manually code the BC: e.g., feed each demo state through the network, output probabilities for actions, and compute cross-entropy loss with the expert's action as the label.
- **Evaluation:** Use metrics like **training accuracy** on the demo set for BC (to see if it memorized those 204 actions). Also, after integration, observe if the initial policy indeed behaves somewhat human-like (maybe run a few games with just the BC model before RL fine-tune).
- **GAIL Tuning:** If using GAIL, leverage the *imitation* library example notebooks ³⁵ ³⁶. Note you'll have to train potentially for many environment steps; fortunately, you can initialize the policy with BC in GAIL too (imitation library does allow that).
- **SQIL** (if off-policy): There's a mention of `imitation.scripts.ingredients.sql` in the docs ³⁷ which hints the library might support SQIL as well. This could be easier than full GAIL – essentially add demo transitions with positive reward into a replay buffer of an off-policy learner like DQN or SAC. If we end up using SAC, we could implement SQIL by manually adding demo experiences to its replay with high reward.

Implementation Guidance

Start by organizing the demo data. From `~/rugs_recordings/` JSONL or the EventStore Parquet, extract sequences of states and the human actions taken. Ideally, reconstruct each game's timeline with flags where the button was pressed. This gives you trajectories: e.g., in game 1, human waited until tick T, pressed BUY, waited, pressed SELL at tick T2, etc. From this, generate training pairs. If using *imitation* library: create `TrajectoryWithRew` objects (though rewards not needed for BC/GAIL). Alternatively, dump a CSV or JSON of state→action mappings.

For **Behavioral Cloning**: you can either use the *imitation* library's `BC` class or do it yourself. If using the library, it might look like:

```

from imitation.algorithms import bc
policy = PPO(...).policy # untrained policy (or a policy network of similar
architecture)
bc_trainer = bc.BC(observation_space=env.observation_space,
action_space=env.action_space, demonstrations=expert_trajs)
bc_trainer.train(n_epochs=100)

```

This will train the policy (the same architecture as PPO's policy) on the expert data. Ensure `expert_trajs` is prepared accordingly (they have utilities to load from rollouts or data). Given we have few demos, might only need a few epochs to fit them (don't overfit to noise though).

After BC, you can load that policy into PPO. SB3 doesn't directly take an initialized policy network easily, but one trick: train a PPO for 0 timesteps to initialize its structure, then manually copy weights from the BC model if needed. If using the same network architecture and library, perhaps easier: use SB3 with the pre-trained weights by setting the model's parameters. Alternatively, since BC and PPO both use the same policy class, you might just do BC within SB3's policy (the code might allow directly optimizing the policy parameters via supervised loss).

Once integrated, run PPO training with a smaller learning rate for a while to refine. If you want to continuously mix IL and RL, consider a **loss add-on**: for example, add to PPO's loss a term like $\beta * \text{cross_entropy}(\pi(a|s), \pi_{\text{expert}}(a|s))$ for states from the demo set. This requires you have expert policy π_{expert} or just the one-hot from demo. Some papers (like *DeepMimic* or others) use this approach to constrain policy to not stray too far. The *imitation* library's BC regularization (as in the Augmenting GAIL with BC idea ³²) could inspire how to implement that.

Remember 204 is small: risk of **overfitting in BC** is high (the network might just memorize those states). To gauge, we could split demos into train/test and see if BC generalizes. But given the very small set, maybe just train on all and accept it might be overfitting – RL fine-tuning with actual reward will correct any misgeneralizations hopefully.

Also, consider **multi-objective IL**: We could incorporate a few of the demonstration trajectories as *behavior shaping* in the reward. For instance, if we know the expert usually cashes out around 30x, we could add a reward bonus when the agent does that in RL training. But this is indirect and possibly redundant if we have direct IL.

All combined, the path likely to yield quick gains: **pretrain with BC -> PPO fine-tune**. This is a well-trodden path (used in e.g. Atari with demonstrations, robotics, etc.). Ensure to evaluate the impact: you can run two training runs with and without BC pretraining to confirm it indeed learns faster or achieves higher final reward.

References

- Jena et al. (2021) “Augmenting GAIL with BC for sample-efficient IL” – highlights that **BC converges quickly but sub-optimally, GAIL is asymptotically good but needs many interactions, and combining them yields the best of both** ²⁷ ²⁸. This supports our approach of BC pretrain + RL (or BC+GAIL).

- *Imitation Learning via Reinforcement Learning with Sparse Rewards* (Reddy et al. 2019, SQIL) – demonstrated that giving expert transitions a fixed high reward and others low can let standard RL learn from demos, outperforming BC and nearing GAIL ³¹. Suggests simpler methods like mixing demo into RL can work well.
- *imitation* library docs – example of using GAIL with expert trajectories to train a policy (CartPole example) ³³ ³⁶. Useful for implementation if we go adversarial route.
- Behavioral cloning discussion – emphasizes the covariate shift problem and need for lots of data or iterative expert queries ³⁸ ³⁹, underlining why pure BC on 204 points is not enough and why we integrate with RL.
- Our own project context (REPLAYER CLAUDE doc) – mentions the synergy: REPLAYER outputs (like trading patterns) were meant to inform RL reward functions ⁴⁰. We are extending this by using actual demo actions to inform the policy directly.

2.5 Feature Engineering for Financial RL

Summary

Feature engineering can make or break an RL trading agent by simplifying the patterns the agent must learn. Our current 36-feature set (expanded to 89 with history, etc.) covers many essential aspects: price, tick count, volatility, positions, PnL, etc. Common **technical indicators** used in trading RL include momentum oscillators (RSI), trend indicators (moving averages, MACD), volatility measures (ATR, Bollinger Bands), and volume indicators ⁴¹ ⁴². In fact, many academic works use a similar number of features (10-20 indicators plus OHLCV data). For example, one study used 14 features including OHLCV and eight technical indicators ⁴³. Our features like `price_velocity`, `acceleration` are akin to momentum indicators, and “sweet_spot” and “sidebet outputs” are custom domain-specific features giving the agent foresight or context. This is great, but we should be cautious of too-high dimensional input relative to sample size – 89 features is a lot. Redundant or noisy features could slow learning or confuse the agent. Techniques like **feature selection** or **dimensionality reduction** can be beneficial. We could apply a simple **correlation analysis** on our dataset to see if some features are collinear or uninformative. If yes, drop or combine them. Alternatively, use an unsupervised method: e.g., train an autoencoder or PCA on the feature set to see if it can be compressed to fewer dimensions without much loss. However, black-box compression might obscure interpretability.

Another angle is using more **advanced network architectures** to handle features. For instance, if we provide raw time-series (like a sequence of recent prices), a 1D CNN or transformer could automatically learn features like moving averages or breakouts. Our agent currently uses manually engineered features plus perhaps a short history. We might consider adding **embedding layers** for certain inputs. For example, “game_id_hash” is mentioned – if this is some categorical identifier, an embedding could represent it (though likely not needed if game dynamics are similar across games). If there are discrete features (like phase of game or flags), those could be one-hot encoded (which they likely are) or embedded if many categories.

Time-series preprocessing: It’s typical to use returns (percentage change) instead of raw prices to stationarize the data. In our game, price always starts ~1 and grows, so maybe not stationary but bounded by nature of game. We already have price change rate (velocity) which serves that purpose. One might also include **moving averages** of price or exponential moving averages as features to capture trend. But given the game’s exponential growth nature, maybe not needed – velocity and acceleration implicitly cover short-

term trend. **Differencing** could be used to remove trend, but here the “trend” is just the upward drift until crash, which is inherent to the game. So perhaps the key is capturing *how fast* and *how long* the price has been rising, which we do.

Market microstructure: In a traditional market, features like order book imbalance, trade volume, etc., can improve performance. In our case, relevant microstructure might be: number of players currently still in (since if many have cashed out, maybe rug odds increase?), or the “rugpool amount vs threshold” which we have (rugpool_ratio). If we can get any info on how other players are acting (maybe from `other_player` events), we could create a feature like “percentage of players exited so far” or “volume of bets still in the game” – those might signal something (e.g., if few players remain, risk might be higher as pot is small? or conversely, if pot huge maybe they want to rug?). If accessible, these could be interesting additions.

We should be mindful of the **curse of dimensionality**: 89 features might be fine if many are low-information, but the agent’s network must figure out which matter. To help, we can either reduce features or guide the network’s focus. Using an **attention mechanism** could let the model weigh which features (or which time steps in a sequence) are important. For instance, a self-attention over the sequence of last N price ticks could let it learn to focus on recent acceleration. Similarly, attention over feature groups might pick out relevant ones in certain contexts (though that’s more exotic). Another approach: **Feature selection via domain knowledge** – e.g., if some features are rarely varying or mostly noise, drop them. We should examine the variance of each feature in our dataset of episodes; features with near-constant value can be removed, and highly correlated ones might be merged.

Finally, consider **successful trading bot features**: Many real trading strategies incorporate indicators like RSI (to measure overbought/oversold), MACD (to gauge trend momentum), volatility indices (like our volatility feature), and sometimes *higher-level features* like pattern recognitions (head-and-shoulders patterns, etc.). We might not need such complex pattern features explicitly if our neural net can learn them, but if we suspect a certain indicator would help (say RSI might be relevant if price growth slows, indicating an upcoming rug?), we can try adding it. Our empirical analysis suggests optimal hold times and risk increase after certain ticks – perhaps include the **temporal rug probability** as a feature (we have something similar in sidebet outputs like ticks_to_rug_norm or is_critical).

In summary, our feature set is already rich, but we should ensure they are **normalized** (volatility, price, etc., on comparable scales) and possibly **prune or combine** where necessary to reduce dimensionality.

Recommendations

- **Normalize and Scale:** Ensure each feature is on a comparable scale (0-1 or standardized). This prevents dominance of large-scale features. For example, tick count goes up to maybe 300; price can go high; these should be normalized (perhaps divide tick by 300, price by an expected max, etc.). Using `VecNormalize` in SB3 can do this automatically per-feature.
- **Feature Importance Analysis:** Once a preliminary model is trained, perform an importance analysis. This could be as simple as checking the magnitude of policy network weights for each feature, or using a permutation importance approach: vary one feature and see impact on action or value. Drop features that seem to have negligible or redundant effect. If training is slow or unstable, try a reduced feature set focusing on the most informative ones (e.g. price, velocity, volatility, sidebet signals, remaining players).

- **Add Technical Indicators if Missing:** Consider adding **Relative Strength Index (RSI)** for the price over short windows (measures speed of gains vs losses), or a **moving average convergence (MACD)** type indicator for price growth. These could help the agent identify when the upward trend is weakening. Our custom features may cover this, but it's worth evaluating.
- **Dimensionality Reduction:** If we suspect 89 dims is too high, use PCA or autoencoder on recorded data to see if a lower dimensional representation (e.g. 10 or 20 dims) captures most variance. You don't necessarily feed PCA features to the agent (losing interpretability), but it might inform which features cluster together. Alternatively, use a smaller network architecture to force the model to implicitly reduce dimensionality (like first layer having fewer neurons than features).
- **Time-Series Features:** Ensure the agent has access to short-term and long-term trends. If our history features (last 5 games stats) aren't too useful for within-game decisions, maybe replace them with within-game recent history (like last 5 tick's price changes). Actually, having features like price velocity and acceleration means we are already summarizing short-term history in two numbers. If that's sufficient, fine. If not, consider explicitly including, say, the price 5 ticks ago or a small window of past prices normalized – or use an LSTM/attention network to handle it.
- **Include Market Context:** If feasible, incorporate features about other players (e.g. how many players have cashed out so far, current rugpool size relative to threshold). These might signal how close the game is to a likely rug event (for instance, if rugpool is near threshold or number of active players is low). We have rugpool_ratio and instarug_count (not sure how instarug works, but if it indicates immediate rug triggers, include it). Make sure the agent can sense *game progression* not just via tick count but via risk accumulation.
- **Regularize Feature Use:** If using an advanced model (like L1 regularization or dropout), it can prevent over-reliance on any one feature and encourage generalization. For example, apply dropout on input features during training so the policy can't latch onto one spurious feature exclusively.
- **Benchmark with Simpler Feature Set:** As a sanity check, try training an agent with a minimal feature set (price, tick, balance, maybe sidebet predictions) and see if it can learn anything. If it can, then each extra feature should justify its inclusion by boosting performance. This helps identify if some engineered features were actually distracting.

Tools/Libraries

- **Pandas/NumPy Analysis:** Use pandas on the Parquet data to compute correlations matrix of features. Look at correlation > 0.9 clusters – you might drop one of each highly correlated pair.
- **ta-lib or pandas-ta:** Libraries that compute technical indicators (RSI, MACD, etc.) easily. We can use these on the price series to generate additional features for our dataset if needed.
- **Feature Plotting:** Plot some features over time for sample games to visually see if they spike or change before a rug. E.g. does volatility or acceleration jump shortly before a crash? If yes, agent should definitely have those. If no, maybe those features aren't predictive and could be noise.
- **sklearn PCA:** Use scikit-learn's PCA to see variance explained by first few components. If first 5 PCs explain 95% variance, our features have a lot of redundancy.
- **Embedding layers:** If using PyTorch for custom policy, you can add `nn.Embedding` for any categorical feature (not sure if we have any truly categorical aside from maybe game phase). Could embed phase of game if it's categorical (like early, mid, late based on tick?). But tick is numeric so likely fine continuous.
- **SHAP or LIME (Explainability):** These are more for supervised models, but you can adapt them to RL by explaining the policy or value function as a "model". SHAP values might tell which features are contributing strongly to decisions. This could be an advanced analysis if needed.

Implementation Guidance

Refine the feature pipeline in the **Observation Builder** (Pipeline D) before feeding to training. Implement normalization: e.g., add fields like `norm_price = price / 100` (if 100x is a typical scale), `norm_tick = tick/300` (since 300 ticks ~79% rug chance). Or simply rely on `VecNormalize` which will do running normalization – that might be easier. If using `VecNormalize`, just make sure to save the normalization stats when saving the model so you can apply them in deployment.

Consider computing some new features: e.g., RSI over last 10 ticks. $RSI = 100 * \text{avg_gain} / (\text{avg_gain} + \text{avg_loss})$ over a window. Given the game's nature, perhaps simpler: compute price change over last 5 ticks (we have velocity = maybe 1-tick change? If velocity is immediate derivative, maybe include a longer-term change). Maybe a feature like `momentum_10 = price_now / price_10_ticks_ago` to capture how much it grew in 10 ticks. That could complement instantaneous velocity with a slightly longer view.

Double-check existing features: *sweet_spot* (3) presumably indicates if price is between 25x and 50x (maybe a binary or one-hot for below/in/above sweet spot). That's useful since it highlights the historically profitable zone. Ensure that is correctly set (it should likely be 1 if in [25,50)x range, 0 otherwise, or something similar). *duration_pred* (4) – presumably sidebet's predicted remaining time normalized and such. Those are already sophisticated features from the side model – definitely keep them as they presumably provide foresight (like probability of rug in next N ticks). The *rug_prediction* (5) features (like probability, confidence, etc.) are extremely valuable – essentially giving the agent an edge if it learns to trust them. Make sure these are updated each tick in the env. They might make some other features redundant (if the side model is accurate, the agent might rely heavily on `should_exit` flag). But that's okay, as long as it doesn't blindly trust a possibly flawed model; hence, keep raw features as well so it can cross-verify.

If computational cost is fine, it's okay to leave many features and let the neural network figure it out. But simpler networks converge faster, so leaning down where possible is good. Perhaps in Phase 1 training (minimal reward), stick to essential features. Later when adding complexity (like ensemble with sidebet), you can add more.

One more idea: **Portfolio features** – since the agent can have multiple positions, features about those positions (quantity, entry price, unrealized PnL) are included. Ensure we include enough info for each position (they said up to 10 positions * 3 features). If the agent rarely opens more than a couple positions, maybe limit to e.g. 5 slots to reduce input size. Or aggregate positions (like total invested, avg entry) which might be easier. But the current design seems fine.

References

- Letian Wang (Medium, 2021) – noted using 14 features (OHLCV + 8 technical indicators) for an RL trading model ⁴³, including common indicators like RSI, CCI, etc., confirming that adding domain indicators is standard.
- Delft Thesis (2020) on Feature Engineering in RL for Forex – found that adding certain technical indicators and doing feature selection improved a DQN's trading performance ⁴⁴. Indicates that not all features are equal; some selection helps.

- FinRL library – uses a default set of technical indicators: MACD, RSI, CCI, DX ⁴⁵, and even a “turbulence index” as a feature (a risk measure of market volatility) ⁴⁶. In our game, volatility and rugpool_ratio play a similar role to turbulence (risk level).
- Srivastava *et al.* (2025) – multi-objective reward paper notes the need for risk measures; by analogy, including risk-oriented features (downside risk, drawdown) is useful ²² ⁴⁷. We have some risk features (balance_at_risk_pct, etc.), which align with that thinking.
- Empirical results from our analysis (from Part 1.5): highlight key thresholds (ticks 138, 69, etc.). We might incorporate those directly as features (e.g., a boolean “danger_zone = tick > 138”). If not already a feature, it could be. This is indirectly present via tick count anyway.

2.6 Model Validation & Deployment

Summary

Validating an RL trading model requires going beyond the training reward. We need to evaluate on actual trading metrics to ensure real-world viability. Key metrics include **total return (ROI)**, **Sharpe ratio** (return vs volatility of returns), **max drawdown** (largest equity drop, indicating risk), and **win rate** (percentage of games or trades ending positive). A model that maximizes reward in training might do so by taking on huge risk (which reward might not fully penalize). So computing risk-adjusted metrics is important. For example, a strategy that doubles money often but occasionally goes bankrupt might have high average reward but unacceptable risk in deployment. Tools like **Quantopian’s Pyfolio** or custom backtesting scripts can generate these metrics and even visual equity curves ⁴⁸ ⁴⁹. We should set aside some historical games as a test set (or use the later portion of timeline for validation). Alternatively, since the environment can simulate new games, we can generate a large number of test simulations (with different random seeds) to evaluate statistical performance. **Overfitting detection** in RL can be subtle: if the model performs significantly better on the exact games it was trained on versus new games, that’s a sign. Also, if the policy seems to have memorized a sequence (which could happen if environment was deterministic or had limited randomness), introducing slight variations and seeing performance drop is a test.

Backtesting in our context is essentially running the agent on past data or simulated episodes without learning and measuring outcomes. It’s crucial to run many episodes to get a stable estimate of win rate or average ROI, due to high variance in individual games (a single rug at an unlucky time can ruin one episode). We might use statistical tests or confidence intervals to assess performance (e.g., 95% confidence that mean ROI is positive). Another check: compare the agent to baseline strategies: e.g. always cash out at 2x, always hold to 50x, random cashout, etc. This gives a benchmark – our agent should beat trivial strategies consistently to be considered successful.

For deployment, **paper trading** (simulated trading on live data) is the recommended intermediate step ⁵⁰ ⁵¹. We connect the agent to the live Rugs.fun game in observation mode – it decides actions but we don’t execute them for real money, we just track what would have happened. This tests it under true live conditions (latency, unpredictable variations) without risk. Only after a period of paper trading with good results should it trade real funds. Additionally, implement **safety constraints**: e.g., limit the bet sizes initially (maybe even lower than the max allowed) until confidence builds, or have a rule that if the agent’s performance drops below a threshold (like loses 3 games in a row badly), it pauses and alerts us.

Deployment considerations: Use the Playwright automation as planned but with oversight. Possibly integrate a **kill switch** – an automated way to immediately exit positions or stop the agent if it behaves

erratically (you can monitor its actions and if something crazy like betting max on tick 1 repeatedly happens, intervene). Logging is also critical in deployment: log every decision, state, and outcome for post-mortem analysis. Another idea: gradually increase real exposure – start with minimal bets (0.001 SOL) even if the agent thinks it can do more, to test its decision-making under real conditions. As it proves profitable, scale up bet sizes carefully.

For **validation/test splits** in training, one approach is **time-based split**: use older game data to train, and the most recent few recorded games as a test. Because markets (or games) can change over time, testing on later data shows if the model generalizes. If we had many games, we could do cross-validation by games (train on some games, test on others). But with limited recorded games, better to rely on simulation tests.

Finally, consider **overfitting detection** during training: one technique is to have a duplicate environment with a different random seed or some variations and evaluate the agent periodically there. If the training reward keeps rising but evaluation reward plateaus or falls, that could indicate overfitting to training scenarios. Regular evaluation callbacks as suggested help spot that.

Recommendations

- **Hold-Out Evaluation:** Reserve a subset of recorded games (or specific random seeds) that the agent never sees during training. After training, run the agent on this **test set** and measure ROI, Sharpe ratio, max drawdown, etc. Only consider the model good if it performs well on unseen games.
- **Backtest on Simulated Data:** Even if using simulation, treat that as backtesting – run the final policy on, say, 1000 simulated games with random outcomes. Compute average profit per game and standard deviation. Look at distribution of outcomes (what % of games lost money, what % doubled money, etc.). This distribution gives insight into risk.
- **Benchmark Against Baselines:** Compare the agent's performance with baseline strategies:
 - *Always Sell Immediately (e.g. at 1.01x)* – very safe, low profit.
 - *Always hold to 50x then sell* – high profit if achieved, but likely to rug often.
 - *Sell at median (e.g. 138 ticks or ~maybe 10x-20x)* – a moderate strategy.
 - *Random Sell Time* – for reference (16.7% win as sidebet model indicates).Ensure our agent significantly outperforms these in ROI and ideally in Sharpe (profit with similar or less risk).
- **Risk Metrics:** Calculate **Sharpe ratio** of the agent's returns over many games ⁴⁸. A positive Sharpe (above 1.0 is great) indicates it's beating risk-free returns per volatility. Also record **max drawdown** – e.g., worst cumulative loss streak. If max drawdown is very large relative to total profit, the agent might be one rug away from disaster, which is a concern. Possibly incorporate *Calmar ratio* (annual return / max drawdown) to evaluate risk-adjusted performance.
- **Overfitting Checks:** If possible, randomize some environment parameters each episode slightly (e.g., vary the rug probability distribution a bit) during training. If the agent still performs well, it's robust. Or train multiple agents with different random seeds and see if they converge similarly – if one wildly overfits a particular sequence, their results will differ.
- **Paper Trading Trial:** Before any real deployment, run the agent in live “dry-run” mode for a sufficient period (maybe 50-100 games) on the actual platform without real bets. Log its decisions and the hypothetical profit. This will catch any integration issues (like timing mismatches, observation lags) and show if it handles real randomness well.
- **Deployment Safeguards:** Enforce basic safety in live trading:
 - Limit maximum bet size to a conservative amount initially, regardless of what the agent decides.

- Possibly have a human review or an automated rule for extreme actions (e.g., if agent wants to do EMERGENCY_EXIT when probability is low, or skip obviously profitable trades, flag that).
- Keep a dashboard of performance in live trading to monitor in real-time (e.g., cumulative profit, recent win/loss streak).
- Plan for continuous learning: If the live environment changes (maybe the game updates its mechanics), be ready to collect new data and retrain or at least validate the model on new data periodically.

Tools/Libraries

- **Pyfolio / QuantStats:** Libraries to analyze trading returns. We can feed the sequence of profits/losses from our agent's backtest into Pyfolio to get a full report (Sharpe, drawdown, monthly returns, etc.) ⁴⁸. QuantStats (a newer lib) can also produce nice stats and plots.
- **Matplotlib/Plotly:** Plot equity curve of the agent over episodes: ideally it's steadily uptrend with some dips at rugs. This visual is helpful to illustrate risk vs reward (big downspikes indicate rugs).
- **Unittest for Strategies:** Write simple test harness comparing strategies. For example, simulate 1000 games with a given fixed strategy (like always sell at 30x) – measure outcomes. Use this to have baseline numbers to beat.
- **Evaluation Script:** Create a script (maybe in `scripts/evaluate_agent.py`) that loads a trained model and runs it on either recorded or simulated games and outputs key metrics. Automate this so it becomes part of model release: every model comes with an evaluation report.
- **Continuous Integration:** If possible, integrate a test in your pipeline that after training, automatically does some evaluation and maybe even rejects the model if it underperforms baseline or some criteria. This ensures only promising models go to deployment.
- **Paper Trading APIs:** The Rugs.fun game likely doesn't have an API, but since we have the Playwright automation, we can modify it to not actually confirm bets or use a test server. If not, perhaps deploy on testnet (if it exists) or with trivial amounts as "paper". For stock trading, Alpaca API and others have paper trading environments ⁵²; for our game, we simulate it ourselves. But concept stands: do not trust the model with money until proven.
- **Logging & Monitoring:** Use a logging library to output agent decisions and outcomes in real-time. Could even integrate with a dashboard (maybe use Plotly Dash or a simple web UI) to watch performance. In deployment, set up alerts (if cumulative loss > X, or if unusual behavior, send a Slack/msg).

Implementation Guidance

Implement a comprehensive `validate_model(model, env, episodes=100)` function. This will run the model for given number of episodes on a provided env (which could be a replay of recorded games or a random sim). Collect per-episode results: profit percentage, whether it went bankrupt, number of trades made, etc. Then compute aggregate metrics.

For Sharpe ratio: Since one "episode" is like one trade sequence, we might treat each episode's outcome as one "return". But Sharpe is normally on a sequence of periodic returns. Alternatively, consider each tick or each small time interval as one period and compute returns. However, easier is to compute an equivalent: for example, treat each game's profit as an independent result and compute mean/std on that. Or compute daily Sharpe if needed (not exactly applicable here). Since our agent resets each game, maybe just use "win rate" and "average ROI per game" more.

Max drawdown: track the agent's equity if it played sequentially with reinvesting. E.g., start with \$100, after each game update equity by profit/loss, track the peak and trough. This will give a drawdown metric across a series of games. This requires deciding an order of games to simulate; random is fine or chronological if using actual data.

To detect overfitting: If you have multiple distinct environments (for instance, maybe a slightly different parameter set for environment, or different random seeds), evaluate on those. If performance is consistent, likely not overfit. If performance only good on the exact training conditions, caution.

Safe deployment: In Playwright, we can implement a wrapper that always checks the agent's decided action against certain conditions. For example, if `action_type == BUY_BOTH` and `bet_size` is max and sidebet predictor says high rug probability, maybe override or log a warning. At least initially, maybe lock out certain high-risk actions (like "EMERGENCY_EXIT" should only be triggered if the sidebet or something agrees). Over time as confidence grows, ease these restrictions.

Keep an eye on **latency** in deployment. The agent must act quickly on rug signals. Ensure the pipeline from WebSocket event to agent decision to action execution is optimized (maybe the reason they planned to integrate with the event bus and state). Validate that in fast games the agent doesn't lag – perhaps test with simulated faster tick speeds. If latency is an issue, consider a simpler model (fewer computations) or some heuristic to pre-compute decisions a few ticks ahead.

Finally, plan a **post-deployment monitoring** strategy: log every game outcome when live, and periodically retrain or fine-tune if needed. The environment is stationary (game rules fixed), but distribution of outcomes could shift if, say, more players join or other external factors. So treat the deployed model as version 1 – monitor and collect new data of its performance, then feed that back into training for version 2 if improvements are seen.

References

- FinRL docs on backtesting – stresses importance of using tools like pyfolio to evaluate strategy performance with Sharpe, annual return, etc., and suggests comparing against benchmark indices ⁴⁸ ⁴⁹ (in our case, compare against baseline strategies).
- LuxAlgo blog "*RL in Market Simulations*" – discusses evaluating RL agents with traditional trading strategy metrics and using walk-forward (rolling) validation to avoid overfitting ⁵³ ⁵⁴ .
- Reddit discussion – highlights that backtesting is for checking mechanics (does agent do what it should) and that real performance needs forward testing ⁵⁵ . Reinforces doing a paper trading phase after backtest.
- Alpaca blog on deploying deep RL for trading – outlines a step-by-step to deploy agent to Alpaca paper trading and then live, with precautions ⁵² ⁵¹ . The approach is analogous to ours.
- Our project docs – mention of a **Visual Replay (REPLAYER)** and analysis tools. We can use REPLAYER to visually verify the agent's behavior by replaying its actions on historical games to spot any glaring mistakes.

2.7 Tools, Libraries, and MCP Servers

Summary

To streamline development and debugging, we should leverage various tools and libraries beyond SB3. **MCP servers** (as mentioned in the context) are likely referring to *Modular Collaborative Programming* servers integrated in our Claude-flow environment (e.g. the ChromaDB vector store is exposed via an MCP for the assistant). Specifically, we have a **ChromaDB MCP server** already ⁵⁶ ⁵⁷, which we use for knowledge retrieval. For ML/RL, are there similar servers? Perhaps connecting a **Jupyter notebook server** as an MCP for live code execution (similar to how we use this environment with Python execution) is useful. There might also be an **evaluation server** or plotting server to allow quick data analysis via the assistant. In our dev stack, we rely on Python libraries for RL (SB3), data (DuckDB, Pandas), etc., so hooking those into a collaborative environment helps.

RL debugging/visualization tools: One handy tool is **TensorBoard**, which we have been using for training curves. Additionally, libraries like **Weights & Biases** (wandb) provide more sophisticated experiment tracking – including hyperparam management, comparisons across runs, and even media (we could log short GIFs of agent gameplay to visually inspect). Another tool, **gymnasium.wrappers.Monitor**, can record videos of episodes. We can use that in our validation runs to generate a video of the agent playing, then watch it to catch odd behaviors. There's also **Scope** from DeepMind or other RL-specific visualizers, but likely not needed here.

Alternatives to SB3: **Ray RLlib** is a robust library that supports distributed training and multi-agent easily. It has integration with Tune for hyperparam search. RLlib could be considered if we want to scale out training to multiple machines or try advanced features (but it might be overkill for now). **CleanRL** is a simpler alternative with very clean, single-file implementations of RL algorithms, great for quick experimentation or customizing an algorithm. There's also **TF-Agents** or **Stable Baselines2** (older) or **OpenAI Baselines** (no longer maintained, SB3 is its successor). Given SB3 serves us well, alternatives would be mainly if we hit a limitation (like wanting multi-agent or something).

Pre-trained models: In trading, there aren't widely available pre-trained RL models because environments differ. But one could use transfer learning from a related game if available. Not applicable here unless maybe we consider using a model trained on a similar crash game or a simpler sub-game (like maybe a model trained to predict rugs could be transferred, but we already have a predictor separate). However, our sidebet predictor *is* a pre-trained model in a sense – we should use it as part of our agent's decision process, which we are doing by feeding it into obs. Another transfer: maybe a model trained with a simpler reward or simpler environment can be used to initialize a more complex training (curriculum). For instance, train an agent to simply survive (avoid bankruptcy) first, then use that as init for full task.

MLOps tools: We definitely want to track experiments. **Weights & Biases** (wandb) is highly recommended for RL. It can automatically log hyperparams, training curves, and let us compare runs. This helps as we do many experiments. SB3 has integration (just pass Monitor wrapper and a Wandb callback or use their SB3 integration script). There's also **MLflow** for tracking, or **TensorBoard** (less collaborative but fine).

For code, since this is a multi-repo project, ensuring version control and coordination is key (which we have via GitHub presumably). Possibly set up a CI pipeline to run tests (they have a test suite of 141 tests as per

REPLAYER doc ⁵⁸). Also, **profiling tools** might be needed if performance is an issue (cProfile for Python to find slow parts in env step or model inference). But given the scale, it's probably fine.

MCP server specifically: - ChromaDB – we have it to query knowledge (like an encyclopedic memory). We might not need to query it during training except if we built an agent that asks for advice (RAG approach), but interestingly they mention RAG for claude-flow. Perhaps not directly relevant to optimizing RL, though one could imagine an agent that calls a “rugs-expert” knowledge base at runtime (but that breaks the closed-loop RL assumption, though a cool concept). - Other MCPs could be hooking into the environment from the assistant side – e.g., an **MCP to run code (like a Python exec server)**, which we effectively have in this environment. Possibly they have a Jupyter MCP (the search result shows `start-jupyter.sh`, `jupyter/CONTEXT.md`), which suggests they might have a persistent Jupyter environment that could be accessed). That would allow interactive analysis or plotting by the assistant. - They might have a **Plot MCP** or similar (less likely unless custom made). - It's possible to integrate a **Matplotlib server** to output charts accessible to the assistant.

Anyway, main point: harness available tools to expedite development: - Use Jupyter or IPython for quick prototyping and analyzing data (we are doing some of that via the assistant environment). - Use a vector store (Chroma) for knowledge retrieval (like storing documentation, research papers so the assistant can query for quick ref – they have started that). - Possibly use **GitHub integration** to fetch code from repos quickly when needed (which we did). - If any cloud compute is needed for training, could integrate with an orchestration (Ray or Slurm), but right now probably local is fine.

Recommendations

- **Experiment Tracking:** Start using Weights & Biases (wandb) for all training runs. It will log hyperparameters, training rewards, and you can add custom metrics (profit, Sharpe, etc.). This makes it easier to compare runs and roll back to best model.
- **Visualization:** Integrate environment video recording for debugging. Wrap the env with `Monitor` to record videos of episodes. After each training or evaluation, review a few gameplay videos to ensure the agent's behavior is sensible (this can catch things like it making an obviously wrong move that metrics might not immediately flag).
- **Alternate Libraries:** If needing to experiment with different algorithms (like Ape-X DQN, or multi-agent scenarios), consider RLlib. It can scale out and has many algorithms implemented. However, keep SB3 as the main pipeline for now since it's stable and well-understood. Only switch if a specific feature is required.
- **Auto Hyperparameter Tuning:** Use Optuna or Ray Tune to automate finding the best hyperparams (we touched this in 2.3, but here as a tool: Ray Tune integrates with SB3 or RLlib to run many trials in parallel). This saves time compared to manual trial-and-error.
- **MCP Integration:** Continue leveraging the ChromaDB knowledge base (maybe expand it with important RL literature or our own notes) so that our AI assistant (Claude) can quickly access domain knowledge when coding or troubleshooting. If a Jupyter MCP is available, use it to run heavy data analysis or model introspection tasks through the assistant interface.
- **Continuous Testing:** Use the existing test suite (141 tests mentioned) to ensure new changes don't break core functionality ⁵⁸. For example, if we modify the environment or reward, run `pytest` to make sure all tests pass. Add new tests for new components (like if we implement a new reward or feature calculation, write a test for it). This will be crucial as complexity grows.

- **Collaboration Tools:** Keep using GitHub for version control and perhaps project boards to track tasks. For documentation, maintain up-to-date markdown files (like those CLAUDE.md) with key decisions and results – these are useful for future maintainers or for reference.
- **Real-Time Monitoring:** Once deployed, use a monitoring tool (could be as simple as a custom script or something like Grafana if we push metrics) to watch the agent's performance. If you have a server running the agent, have it output stats to a dashboard. This falls under MLOps – treating the trading bot like a production ML service.

Tools/Libraries

- **Weights & Biases:** We've discussed but to specify: wandb can be set up by `wandb.init(project="rugs-rl")` and SB3 can log to it through a callback or by logging metrics manually. There's also a wandb SB3 integration example.
- **Ray Tune:** Provides a powerful experiment launcher; e.g., you can define a search space for hyperparams and it will schedule multiple SB3 training processes with different configs. It can use Bayesian optimization to converge on the best results. Use this when we want to fine-tune hyperparams thoroughly (maybe in Phase 1 hyperparam tuning step).
- **Gymnasium Monitor & imageio:** For video. After running some episodes, retrieve the video and watch it. Possibly embed some in reports (though not needed for code, it's for us to debug).
- **ChromaDB:** Already in use, ensure our important project knowledge (plans, analysis results, known best hyperparams, etc.) are indexed. This might indirectly help when using the Claude-flow assistant to answer questions (like it did now).
- **MCP for Jupyter:** If available, it means the assistant can run code directly (which we're doing via the Python tool here). If not already done, consider setting up an IPython kernel that can be invoked via chat to quickly test ideas or plot charts. This is extremely useful for a research agent to validate things on the fly.
- **Hardware Utilization:** If training is slow, consider using GPU (SB3 can use GPU for neural nets via PyTorch if available). Also, if wanting to parallelize beyond one machine, Ray or a simple custom MPI approach could be tried. But likely a single high-end PC is enough for our scale (observations are small, environment is not heavy graphically).

Implementation Guidance

Set up wandb: in the training script, include:

```
import wandb
from stable_baselines3.common.callbacks import WandbCallback
wandb.init(project="rugs_bot", config={...hyperparams...})
model.learn(total_timesteps=..., callback=WandbCallback())
```

WandbCallback will log reward and more. Add custom metrics by hooking into the `on_step` or `on_episode_end` to log profit or Sharpe for that episode. This will give a live view of how actual profit is trending, not just reward.

For debugging, implement a `render()` in the environment (if not already) that maybe prints or visualizes state. Could just print key info each tick when debugging. Use it if something's off to trace through an episode.

Since the question specifically mentions MCP servers for ML/RL dev: possibly they're hinting at adding more AI assistant "superpowers". For example, an **MCP server for plotting** where the assistant can directly ask to plot a series. This might not be set up yet but could be considered. However, given the context, likely they just want to ensure we use the full palette of tools.

One interesting idea: Use an **analysis assistant** (like an offline version of ourselves) to examine the model. For example, after training, take the model's policy network and run some analysis – like see how output changes with certain feature changes (sensitivity analysis). This could be partially automated.

In summary, make heavy use of experiment tracking, keep the knowledge base updated, test thoroughly, and use accessible frameworks (like SB3, RLlib, etc.) to avoid reinventing wheels. Each problem (hyperparam tuning, multi-env, etc.) likely has a known tool or library – use them rather than writing from scratch.

References

- Project CLAUDE docs – highlight integration of ChromaDB and mention *Claude superpowers* for code and knowledge ⁵⁹ ⁵⁷. This implies our environment is set up for an AI-in-the-loop development, which we should exploit by storing knowledge and letting the assistant help.
- SB3 documentation – details integration with third-party tools (TensorBoard, wandb).
- Ray documentation – shows how to use RLlib and Tune for hyperparam tuning and scaling RL training.
- GitHub CI for RL projects – e.g., Stable Baselines3's repo uses tests and style checks; we might emulate that to catch issues early.

2.8 Advanced Techniques to Consider

Summary

As we refine the system, several advanced techniques from recent research could be considered for potential gains:

- **Transformers/Attention in RL:** The *Decision Transformer (DT)* is a novel approach that reframes RL as a sequence modeling problem, using transformer architectures to output actions based on the trajectory of past states, actions, and a desired return ⁶⁰. In trading, transformers could capture long-range dependencies or seasonality that an LSTM might miss. For example, a DT could conceivably condition on the target return (say, aim for a 300% return) and generate a sequence of actions to achieve it. However, DT typically requires lots of offline data. With our limited demos, a DT might not be trainable from scratch. Another application of attention is within a policy network: using self-attention on a window of recent observations, which might better capture patterns than a fixed-size recurrent memory. There's also *Trajectory Transformer*, etc., but all are data-hungry. If we significantly expand our dataset (or augment with simulated trajectories), trying a Decision Transformer fine-tuned on our environment could be a cutting-edge experiment. Notably, some recent work attempts to adapt pre-trained language models to RL via Decision Transformers ⁶¹, meaning one could leverage a big pre-trained model (though it's unclear how that helps with our specific game without relevant data).

- **Offline RL (CQL, IQL):** We do have a dataset of human (and possibly bot) play. Offline RL algorithms like **CQL (Conservative Q-Learning)** and **IQL (Implicit Q-Learning)** are designed to train from fixed datasets without additional exploration, by addressing distributional shift issues (they penalize Q-values for unseen actions to avoid overestimation). If we had more recorded games (say thousands with varied strategies), offline RL could learn a policy that is safe and reasonable, which we can then fine-tune online. CQL, for example, could take our 929 game sessions as input and output a policy that doesn't go beyond what's successful in the data. This might yield a more stable starting policy than random. It's somewhat related to IL but more flexible (it can surpass the demonstrators by recombining experience). If we feel our environment is close enough to the historical data distribution, we can attempt offline training with something like D4RL methods. Implementation-wise, there are libraries (e.g., d3rlpy in Python) that implement CQL, IQL easily on datasets.
- **Multi-Agent RL:** In Rugs.fun, multiple players play simultaneously, though our bot is focusing on a single player's perspective. There might be an interplay: e.g., if many players cash out early, maybe the game adjusts or the risk changes. A multi-agent approach could simulate other players (maybe a population of agents, including our bot, all learning). This is complex and likely unnecessary if the environment (price growth and rug) doesn't directly depend on others' actions. But if, for instance, rugs were somehow influenced by the pot size or number of players remaining, a multi-agent training where our agent learns in presence of others could yield a more robust strategy. Another angle: train an agent that *predicts other players' behavior* or takes advantage of it. However, unless we identify clear multi-agent dynamics, this may not be worth the complexity right now.
- **Non-stationarity handling:** Markets can change, and even in our game, perhaps the statistical properties might drift (maybe the devs change rug probability distribution). Techniques to handle non-stationarity include **online learning** (continually update the model with new data) and **meta-learning** (learn how to learn, so the agent adapts quickly to new conditions). One practical approach is a **sliding window retraining** – only use the last N games for training or give more weight to recent data, so the agent adapts if things change. Another approach: train a **population of agents** with different assumptions and switch between them based on performance (portfolio of policies). For example, one policy for volatile periods, one for stable – detect the regime and use appropriate policy. This might be overkill for our game unless there are distinct regimes.
- **Uncertainty quantification:** In trading, knowing **confidence** in decisions is valuable. Our sidebet model already provides a form of this (confidence, probability). We could similarly have the RL agent output not just an action but a confidence or distribution over outcomes. Techniques include **Bayesian deep learning** (like dropout as Bayesian approximation to get uncertainty), or training an ensemble of policies and seeing variance in their actions as a measure of uncertainty. Also, **distributional RL** (e.g., QR-DQN) learns a distribution of returns instead of a single expected value – that could highlight tail risks. If we apply distributional RL (like a distributional critic in an actor-critic algorithm), the agent might better account for risk of ruin vs potential gain. Another method: incorporate an *uncertainty penalty* – if the agent is uncertain about the outcome, it might take a safer action. For instance, using the sidebet's confidence: if sidebet says outcome is very uncertain, agent might reduce bet size or avoid trading. Calibrating such behavior can improve risk management.
- **Ensemble of sidebet + RL:** Currently we feed the sidebet predictions into the RL observation. Another way is to have two separate models that vote or are combined via an ensemble. For example, the sidebet predictor could directly decide when to emergency exit (if prob ≥ 0.5 , force

exit), while the RL handles other parts (entry timing, bet sizing). This is like a rule-based override from the side model. We can formalize this as a **hierarchical policy**: top-level decides *whether to be in the game or not (using sidebet's risk)* and lower-level decides *how to trade when in game*. Implementation: treat sidebet model as providing a hard constraint – e.g., if `should_exit=1` from sidebet, override the RL action to sell (or at least strongly bias it). Another approach is an ensemble where both models propose an action and we pick the one with higher confidence. This might improve safety (sidebet acts as a guardian against catastrophic rug). Essentially, by **ensembling**, we ensure that if either model detects high risk, the agent acts on it. Or conversely, sidebet might sometimes be too cautious; maybe have a parameter to weigh its advice.

- **Curriculum learning**: Mentioned earlier but indeed an advanced training strategy – we could start training the agent on easier versions of the game and progressively increase difficulty. For example, initially train in an environment variant where rugs happen later on average (giving the agent a chance to realize profit more often), then gradually shorten the expected rug time back to real levels. This way the agent first learns to trade in a forgiving environment and then adapts to harsher ones. Curriculum could also mean restricting action space at first (maybe only allow buy/sell, no partial or side bets) and then adding complexity. This staged learning can sometimes yield better final performance than learning everything at once.
- **Research papers to explore**: Some relevant ones might be:
 - **“Deep Reinforcement Learning for Automated Stock Trading”** (many such papers) – often combine CNN/LSTM with RL, could have ideas like attention mechanisms on time series.
 - **“Episodic Memory in RL”** – maybe the agent could benefit from remembering outcomes of past games (like meta-learning across episodes).
 - **“Ensemble Agent for Risk Management”** – possibly research on combining signals or agents for trading.

Given priorities, these advanced methods are “cherry on top” once basics work. It’s wise to get the simple PPO agent profitable with minimal reward first. Then, advanced techniques like transformers or offline RL could be explored to push performance further or solve any remaining issues (like if PPO plateaus).

Recommendations

- **Decision Transformer Trial**: If we accumulate a sizable dataset of gameplay (e.g. by running a random or semi-trained agent to generate lots of trajectories), try training a Decision Transformer. This would be a research project on its own – you’d need to adapt the state/action representation for the transformer and condition on return (maybe desired return = some multiple of initial bet). This could potentially learn a strategy from offline data and then we can fine-tune it online. Keep expectations tempered given data requirements.
- **Offline RL for Warm Start**: Use d3rlpy or similar to run **CQL** on the recorded dataset. Even if it’s only ~59 distinct games, CQL might glean some policy (though 59 is very low for offline RL too). If we manage to simulate more games with a mix of random and sidebet strategies, we can produce a larger offline dataset for CQL. A conservative Q-learning approach could ensure the learned policy doesn’t try too radical actions not seen in data, hence likely avoiding total ruin. That policy can then be fine-tuned with PPO to refine.

- **Multi-Agent if Needed:** Only pursue multi-agent simulation if we see evidence that other agents' behavior affects our optimal policy significantly. If so, consider training in a environment with multiple learning agents (perhaps using RLib's multi-agent support). Alternatively, simulate other players with a fixed strategy (like average player behavior) while training our agent – this way, our agent learns to react to others. For now, keep this as a low priority unless game dynamics prove dependent on others.
- **Meta-learning:** Down the line, test if the agent trained in one market condition handles a changed condition. For example, if the rug distribution shifts (say median moves from 138 ticks to 100 ticks), does it adapt? If not, consider meta-RL algorithms (like RL² or MAML) that train the agent to adapt quickly to new conditions by learning an internal model of dynamics. This is complex, so only if we foresee environment changes.
- **Uncertainty & Risk:** Implement **policy ensembles** – e.g., train 3 agents with different random seeds or slight variations, then ensemble their decisions (vote or average their Q-values if using Q). This can reduce variance in decisions and provide a confidence (if all three agree, high confidence; if they diverge, uncertainty). Also consider **distillation** of ensemble into one policy for efficiency later. Additionally, utilize the sidebet's probability as an input to penalize risky actions: e.g., multiply the reward of any action taken while `rug_prob` is high by some factor <1 to discourage betting when risk is high. Or incorporate CVaR (Conditional Value at Risk) in the reward – optimize not just expected value but a risk-adjusted metric.
- **Hierarchical policy (when vs what):** As asked, separating *when to act* from *what action to take*. We could train a high-level model that at each tick decides whether to act or not, and a low-level model that given the command to act decides whether to buy or sell and how much. One way to do this: restrict the action space of the main PPO to just two actions: {No-op, "Invoke Trader"}. If "Invoke Trader", then use a secondary policy or even a heuristic to execute trade (like buy if no position, sell if holding, etc., or a more complex second agent). However, training this hierarchy is non-trivial because the sub-policy and master policy interact. A simpler approach might be to encode this logic in the agent's network architecture: have one part of the network decide action type (including maybe a no-op vs act classification) and another part decide magnitudes. This is already somewhat the case with MultiDiscrete, but we could enforce a structure (like the network first outputs a probability of doing anything vs nothing, then conditionally outputs what to do).
- **Sidebet Ensemble:** Use the sidebet model in a more explicit ensemble. For instance, during each decision, if sidebet's `should_exit=1`, override any action to SELL (emergency exit). Or if sidebet's rug probability is below 0.2, maybe allow the agent to be more aggressive. Basically, encode some rules from the side model as safety checks. Over time, if the RL gets good, you can relax these constraints. This ensures early on the agent doesn't ignore a proven good predictor. It's like an ensemble where sidebet model is a specialist in predicting crashes and the RL is a specialist in maximizing profit; use each for what they're best at.
- **Continuous Learning Pipeline:** For deployment, consider a pipeline where new data (game outcomes, agent's experience) continuously flows into a training process (could be offline updates or periodic online updates). This way the model can keep learning and improving with real experience (especially if environment dynamics shift or if the agent finds new edge cases). Be careful though: online updates can also destabilize a working strategy if not monitored (this becomes a whole ML ops issue itself).

References

- Chen et al. (2021) “*Decision Transformer*” – introduced the concept of using a transformer for offline RL, matching conditional trajectories to desired returns ⁶⁰. Suggests potential for sequence modeling in RL.
- Kumar et al. (2020) “*Conservative Q-Learning*” – key offline RL paper that prevents overestimation on unseen actions, enabling learning from static datasets. Could be applicable if we have enough demo/sim data.
- Kalashnikov et al. (2018) “*QT-Opt*” – an example of an RL that was pre-trained offline on a large dataset then fine-tuned online for robotic grasping; analogous to what we might do with CQL + PPO.
- OpenAI’s “*Multi-Agent Competitive Self-Play*” – though in games like hide-and-seek, illustrates how multi-agent training yields emergent strategies; mentioned as an inspiration if multi-agent aspect considered, but probably beyond scope here.
- Moerland et al. (2023) “*Model-Based RL in Finance*” – not directly asked, but an advanced idea: if needed, we could incorporate model-based approaches (predict price movements and plan actions). We have a model (sidebet predictor) which is a bit like a model of the environment’s risk. Could integrate that into planning (e.g., use Monte Carlo simulation of outcomes to choose best action). This is advanced but could further leverage sidebet predictions.

Part 3: Addressing Immediate and Specific Questions

3.1 Immediate Problems:

- *Reward Hacking*: To ensure the model doesn’t find new exploits, we’ve minimized the reward function and tested it for obvious loopholes. Going forward, we will **continuously monitor the agent’s behavior** during training (via action distribution and performance metrics). If we see signs of new hacks (e.g., suddenly the agent stops trading or spams some action with no economic sense), we will pause and investigate. We’ll also incorporate some of the detection ideas discussed (like anomaly detection or tripwire conditions). Crucially, we’ll maintain the principle of *reward simplicity*: only slightly add complexity once we’re confident the agent truly trades profitably. Each time we add a component (say a small time penalty or so), we’ll re-test for hacking. Essentially, **tight iterative testing and incremental reward design** will catch hacks early. By also validating on actual profit (which is hard to “game” without real success), we ensure that even if the reward is exploited, it won’t show good real-world metrics, alerting us to the issue. In summary, keeping the reward function aligned and simple is our best guard, plus diligent testing ^{12 21}.
- *Sparse Data for IL*: 204 demos is quite limited. We will augment this through **data augmentation and synthetic data**. One plan is to use the sidebet model or a simple heuristic to generate additional demonstration trajectories (effectively an AI or rule-based demonstrator). For example, a strategy that bets small and cashes out at 30x consistently – run that for many games to get trajectories. These may not be optimal, but provide additional training data. We’ll label them as “expert” for IL purposes to bulk up the dataset. Also, as mentioned, we can break each demonstration into multiple labeled decision points (not just the moments of action). By doing so, we expand the supervised dataset. With these techniques, we hope to get a few thousand state-action pairs at least. If that’s still not enough, we might rely more on RL with the demos just as a slight guide (via pretraining or regularization). Another concept: use **transfer learning** – perhaps pretrain the model on a simpler game with more data (maybe a simulation of Rugs.fun with easier

settings or even a different crash game data that might be available) and fine-tune on our data. This could give a better start. Overall, while 204 is small, by creative augmentation and combining IL with RL (so the agent learns from its own experience too), we believe it's sufficient to significantly kickstart learning. We'll be careful not to overfit those 204 points – using early stopping in BC training and perhaps cross-validation on them (if we hold out some demo points to test BC generalization). If needed, we'll ask some human experts to generate a few more demonstrations in critical scenarios to cover gaps.

- *Temporal Credit Assignment*: This is indeed a challenge since an action (buy) may only pay off much later (sell or rug outcome). PPO with a reward discount of 0.995 will assign credit over long horizons, but if that's not enough, we have a few strategies: (1) **Shaping**: We can give intermediate feedback for being “on track” (e.g., unrealized PnL could be given as a signal). Currently, our financial reward is realized PnL, which only comes at sell time. We could augment it by a small **unrealized PnL reward each tick** (the change in mark-to-market value) so that the agent gets a sense of progress. This is effectively like having a continuously valued portfolio – many trading RL setups reward the agent by the change in equity each step ⁶². That way, if an action leads to a big unrealized gain a few ticks later, the agent already sees a reward bump (even if it hasn't sold yet). (2) **Eligibility Traces**: although PPO doesn't natively use them, we could implement a form of n-step return or use Generalized Advantage Estimation (GAE, which we are using) to smoothly propagate rewards back. GAE with a high lambda (0.95 or 0.99) will make the credit assignment more spread out. We'll ensure we use GAE (SB3's PPO does by default) to help with credit assignment. (3) **Model-based hints**: If needed, use the sidebet predictor's expected time-to-rug as a feature – the agent can then anticipate how far future rewards might be. (4) **Reward delay handling**: We must also make sure to **not truncate episodes prematurely**; an episode ends only at rug or manual exit. If episodes were cut at a time limit, credit assignment would break – but we've set it up properly. In summary, combining shaping (like giving profit as it grows) and using PPO's advantage estimation will address delayed reward. And if we see that the agent still struggles to connect the dots (like it doesn't learn to sell before rug because the negative reward comes too late), we might explicitly provide a “pre-rug exit” reward: a small bonus specifically when an exit happens shortly before a rug, effectively highlighting that crucial timing (though careful to avoid misuse).
- *Distribution Shift (Sim to Real)*: We trained on recorded data and our simulator, but live conditions might differ. To combat this, we will implement a **feedback retraining loop**. We'll treat the first phase of deployment as gathering new data: run the bot in paper trading or with tiny stakes and record those games. Then compare distribution of key variables (multipliers reached, volatility patterns) to our training data. If there are differences, we'll update the environment model. For example, if we notice rugs happen faster live than in our old data, we'll adjust the environment's rug probability curve to match current reality, and retrain or fine-tune the agent on it. We can also use techniques like **domain randomization** during training – intentionally vary the rug distribution or other parameters within plausible ranges so the agent doesn't overfit to a narrow set. This way, if reality is slightly different, the agent still copes. Another safety net is the sidebet model: since it's trained on real data and continues to get updated with new data (assuming we update it), it will provide the agent guidance even if the environment shifts. We also plan to keep the model updated: periodically retrain the RL agent on recent games (this could be monthly or after major changes). Essentially, we acknowledge distribution shift risk and counter it by **continuously learning** and ensuring our simulation stays true to live data (our use of real recorded data as basis helps here).

Finally, having conservative risk measures helps – even if distribution shifts, a cautious agent (that doesn't bet the farm on assumptions) will survive and can adapt.

- *Risk Management (When NOT to trade):* Teaching an agent to sometimes do nothing is challenging because we typically reward action outcomes. We have a discrete action "SKIP" or "WAIT" – we need to ensure the agent doesn't get penalized for using it. With our current reward (pure PnL), doing nothing neither gains nor loses, so in theory an agent that finds no profitable opportunity should be indifferent to waiting (which is good). But to further encourage not trading when it's bad, we can incorporate a **small transaction cost or penalty for losing trades**. For instance, already losing money will reflect in negative reward, so it learns not to trade when chance of losing is high. We can also add a tiny penalty for each action (except wait) – this will naturally make it prefer inaction unless an action is clearly beneficial to overcome the tiny cost ²¹. This is akin to discouraging overtrading. In real trading, transaction costs and fees serve this role; we can simulate a small "commission". Another approach is to reward **inaction in dangerous times** via shaping: if sidebet says rug probability > 0.4 and the agent has no position, that's actually a good choice – we might give a small positive reward for "proper avoidance". But this could be tricky and might introduce a hack (agent could just always avoid). More straightforward: design the training scenarios such that sometimes the best action is indeed to do nothing – the agent will learn by comparison that trading in those scenarios yields worse reward than sitting out. We can ensure some games in training have extremely high risk from start (so the optimal is not to play). The agent seeing that any trade in those games leads to loss will learn to skip in those cases. This is a form of **negative example** training. Summarily, our plan is to incorporate either a minor action penalty or use natural losses as lessons so that the agent learns that sometimes zero is better than negative (thus not trading is better than trading poorly). And when deploying, we will keep an eye on its trade frequency – if it's trading when it shouldn't, we might tighten the rules (like only allow X trades per game or require a certain confidence threshold to act, effectively gating actions by some criteria until we're comfortable).

3.2 Architecture Questions:

- *Recurrent vs. Feedforward:* We touched on this – using an LSTM policy could allow the agent to internalize more temporal context than our explicit features provide. This might be beneficial especially if subtle timing patterns aren't fully captured by current features. For example, an LSTM could in principle learn an increasing hazard rate (rising rug risk) by observing a long sequence of no-rug ticks. If we feed just tick count and hazard as features, it knows it in a way, but the LSTM might learn nuances. We will likely experiment with a recurrent policy once the basic feedforward policy is working. SB3 makes it relatively easy to switch to a RecurrentPPO with an LSTM layer. The downside is sample efficiency (LSTMs have more parameters and are harder to train). But if our agent with feedforward seems to struggle with timing or forgets earlier parts of the game, that's a sign to move to LSTM. Another hybrid approach: keep feedforward but increase the history window in features (e.g., include last 5 tick prices explicitly). That might approximate what an LSTM would do, albeit with fixed window. In either case, the aim is to capture temporal dependencies. We should definitely avoid too short-sighted a network. So, yes, if necessary we'll use an LSTM or even an Attention mechanism (which can capture long range without the recurrency issues). The earlier-cited study ⁶ showed an LSTM waited longer and was more patient than a feedforward, implying it recognized longer-term patterns. In trading, patience can be key, so an LSTM could imbue that quality.

- *Decision Transformer*: As said, it's an intriguing idea, but given our current resource (data) constraints, it might be something to consider after we have more experience data. If we do manage to generate a lot of simulated data or incorporate more human play data, a Decision Transformer could be tried to see if it learns better from the offline dataset than PPO from scratch. It could potentially handle the credit assignment by conditioning on target returns – essentially telling it “aim for 300% ROI” might implicitly train it to hold through the sweet spot and exit appropriately. Some recent work even fine-tunes large language models as decision transformers for finance ⁶¹, though that's bleeding-edge. For now, we'll note it as a future experiment. The safer advanced technique is offline RL as discussed, which is more direct.
- *Separate When vs What Models*: This concept can be implemented as a hierarchical policy or gating mechanism. We think introducing an explicit “should I act?” classifier could help, because the space of “when to act” is somewhat separable from “what action to do”. One plan: train a simple supervised model (like a binary classifier using our features) on the demonstration data to predict whether the human acted at a given tick or not. That model essentially learns the conditions under which an expert deemed it was time to push a button. If it's reasonably accurate, we can use it in the agent as a gating function: only allow the RL policy to choose a non-wait action when the classifier predicts it's a good timing (above some confidence). This creates a semi-rule-based approach that could stabilize training (reducing the exploration in clearly bad times). It's akin to an “attention mask” on time steps. Alternatively, we incorporate this into the network architecture: have the network output two things – a probability of acting now vs waiting, and the action details if acting. We then could train that with appropriate losses (e.g., imitate the human's decision of when to act as auxiliary loss). This multi-head architecture might separate the concerns internally. We'll likely try the simpler gating first (with something like the sidebet model or an imitation model as the gate). As we gain confidence, we might relax it to let RL override if it strongly disagrees, but early on it could prevent silly frequent trades. So yes, conceptually separating when and what is sound and we will incorporate that either via architecture or via ensemble (classifier + RL).
- *Ensemble sidebet with RL*: As described, we plan to use sidebet model as an ensemble partner. Concretely, one method is **action veto/override**: if sidebet says “should_exit=True (prob >= 0.4 threshold)”, we force the agent's action to SELL (or at least if the agent wasn't already planning to). We can implement this easily in the trading loop. We might start with a softer approach: incorporate the sidebet's recommendation as a very large positive or negative reward at that tick to incentivize the agent to learn the same behavior. But since we already input `should_exit` feature, the agent might learn it naturally. If not, an override ensures safety. Similarly, if sidebet says “emergency (prob>=0.5)”, absolutely get out. On the flip side, if sidebet indicates low rug probability, we might prevent the agent from doing panicky sells too early. But that we have to be careful – the agent might see another reason to exit (like hitting 50x target) which is valid even if rug probability is low at that moment. So we won't override exits if they are profit-target based. We'll mostly use sidebet for risk avoidance (kind of like a safety layer). This ensemble approach is essentially a form of **safe RL**, where a known safe policy (sidebet exit strategy) shields the learning agent from catastrophic failures. Over time, if the RL becomes very good, we can test turning off the overrides to see if it still performs well (maybe it learned them anyway). Another ensemble approach is average their action preferences, but since sidebet's output isn't exactly an action (it's a prediction), the more straightforward approach is the rule-based integration.

3.3 Training Questions:

- *Learning Rate Schedule:* We will likely adopt a **linear decay** of learning rate, as mentioned, since it's a simple and proven schedule in PPO. We may also experiment with **adaptive LR** (e.g., reduce LR if the training reward hasn't improved in X steps, similar to plateau scheduling). For financial tasks, often a smaller constant LR is used to fine-tune at the end. One strategy: start with $3e-4$ for first 100k steps, then lower to $1e-4$ for stability for another 200k, then even $5e-5$ for final fine-tuning. This could help not overshoot optimum. The exact schedule we'll tune based on observed training stability. If we see oscillations or divergence, decreasing LR might help. Also using **gradient clipping** (which PPO does inherently by clipping updates, but additionally we can clip grad norms) ensures no giant step in policy space.
- *Curriculum Learning:* We are very open to using curriculum. One idea: begin training with an environment variant where rugs are less frequent (e.g., multiply all rug probabilities by 0.5 so games last longer on average). In these easier games, the agent can learn to accumulate profit without being cut short as often. Once it learns to get profit, we up the difficulty (increase rug frequency gradually to normal levels). This way it won't get discouraged by too many sudden losses early. Another curriculum aspect: initially limit the action space – for example, disallow side bets and partial sells in the early training, have the agent just learn to do full buy and full sell. Once it masters that (makes profit), we introduce partial sells or side bets for fine-tuning. This avoids overwhelming it with a huge action space initially. We can implement that by modifying the environment to ignore those actions or by shaping rewards such that those additional actions have no benefit until a later stage. Curriculum could also be based on starting conditions: maybe start with giving the agent a small profit already (like start games at 2x price, so they have a cushion), then remove that cushion later. We need to be careful that curriculum doesn't alter the nature too much or we might teach policies that don't translate. But gradually increasing difficulty is a standard approach (like how you might train a gamer with easier levels first). We'll design a few phases and transitions triggered either by epoch count or performance (like "if agent achieves X ROI in easy env, switch to harder env").
- *Prioritized Experience Replay:* PER is typically for off-policy algorithms. In our PPO setting, we don't have a replay buffer per se – it's on-policy. If we switch to an off-policy method like SAC or DQN, then PER can be useful: it would sample more from rare but important experiences (e.g., experiences around rug events or big wins). If we find that the agent isn't learning enough from those critical moments because they are rare, and if we use off-policy, we will definitely implement PER to amplify them. For PPO, one could approximate PER by weighting certain trajectory segments more in the loss (not common though). Another idea: since we store episodes (we can store them after training), we could create a pseudo-replay for PPO just for analysis: e.g., deliberately feed in past "failure" episodes occasionally to remind the agent. That breaks on-policy assumption though. So likely, PER will come into play if we adopt a DQN or SAC approach for e.g. side bet decisions or something. In summary, PER is valuable if we have off-policy training; for PPO, not directly applicable.
- *Parallel Environments:* We will use as many as feasible given CPU. Likely start with 8, could go to 16 if needed. More envs = more throughput but diminishing returns beyond a point. Also, if using too many, each gets fewer updates per timeframe which can affect learning (but total steps stays same). We'll profile – if environment is lightweight, 16 threads might be fine on a decent CPU. If CPU becomes bottleneck, maybe 8 is optimal. We'll certainly incorporate that from the get-go because it's

a trivial speedup and stabilizer (with parallel envs, the collected batch is more diverse, reducing variance of gradient). If we eventually use an LSTM policy, we might reduce env count or sequence length to accommodate memory. But we'll adjust accordingly.

Lastly, one training consideration: **evaluation frequency** – we will periodically test the policy on greedy execution (no randomness) to see actual performance while training. This early stopping criteria or model selection from those evaluations ensures we pick the best iteration to deploy, not just final.

All these measures combined – careful LR scheduling, curriculum to guide learning, possibly PER if using off-policy, and sufficient parallelism – should lead to an efficient training regimen that converges to a good policy without wasting too many resources or time.

References (for Part 3 answers)

(These have mostly been integrated above in context-specific citations for brevity, drawing from earlier references)

- Handmann et al. (2020) – noted differences in LSTM vs feedforward agent behavior (patience) ⁶ .
- Lilian Weng (2024) – reward hacking mitigation through proper reward design ¹² .
- Srivastava et al. (2025) – multi-objective reward to avoid over-optimization of single metric ²¹ (justifying small penalties/regularization).
- Emergent Mind (2025) – discusses avoiding reward hacking with constraints and oversight (aligns with our ensemble override idea) ¹⁷ ¹⁸ .
- FinRL and other trading RL sources – using transaction cost in env (we will similarly use a small action penalty) ³ .
- Sutton & Barto – general RL textbook insights: eligibility traces for credit assignment, etc., which underpin our GAE usage.

Conclusion: With the above research-backed strategies, we will incrementally build up our trading bot's capabilities while carefully validating each step. Starting from a solid foundation (clean data, simple rewards, and basic PPO), we'll integrate imitation learning to leverage human insight, apply state-of-the-art training tricks (parallelism, tuning, curriculum) to accelerate learning, and use advanced techniques (risk-aware adjustments, possibly offline pretraining) to polish performance. Throughout, rigorous validation with real-world metrics and safety checks will be our compass, ensuring the final deployed agent is not just reward-optimal but truly profit-optimal and robust to the unpredictable nature of the Rugs.fun game.

¹ ² ³ ⁷ ⁸ ²³ ²⁴ ²⁵ ⁴⁵ ⁴⁶ ⁴⁸ ⁴⁹ Single Stock Trading — FinRL 0.3.1 documentation

<https://finrl.readthedocs.io/en/latest/tutorial/Introduction/SingleStockTrading.html>

⁴ Handling Time Limits - Gymnasium Documentation

https://gymnasium.farama.org/tutorials/gymnasium_basics/handling_time_limits/

⁵ How does an episode end in OpenAI Gym's "MountainCar-v0 ...

<https://ai.stackexchange.com/questions/22548/how-does-an-episode-end-in-openai-gyms-mountaincar-v0-environment>

⁶ handmann.net

<https://www.handmann.net/pdf/JRFM-ZenHan2020.pdf>

- 9 DI-engine/dizoo/gym_anytrading/envs/README.md at main - GitHub
https://github.com/opencv/dl-engine/blob/main/dizoo/gym_anytrading/envs/README.md
- 10 gym-anytrading - Package Hub
<https://package-hub.github.io/python/openai-gym/reinforcement-learning/2023/03/24/github-com-aminhp-gym-anytrading.html>
- 11 12 13 14 15 16 Reward Hacking in Reinforcement Learning | Lil'Log
<https://lilianweng.github.io/posts/2024-11-28-reward-hacking/>
- 17 18 Avoiding Reward Hacking
<https://www.emergentmind.com/topics/avoiding-reward-hacking>
- 19 Transcript: Concrete Problems in AI Safety with Dario Amodei and ...
<https://futureoflife.org/uncategorized/transcript-concrete-problems-ai-safety-dario-amodei-seth-baum/>
- 20 Concrete Problems in AI Safety
<https://www.summarizepaper.com/en/arxiv-id/1606.06565v1/>
- 21 22 47 Risk-Aware Reinforcement Learning Reward for Financial Trading
<https://arxiv.org/html/2506.04358v1>
- 26 Reinforcement Learning Tips and Tricks - Stable Baselines
https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html
- 27 28 29 30 32 38 39 proceedings.mlr.press
<https://proceedings.mlr.press/v155/jena21a/jena21a.pdf>
- 31 Imitation Learning via Reinforcement Learning with Sparse Rewards
<https://arxiv.org/abs/1905.11108>
- 33 34 35 36 37 Generative Adversarial Imitation Learning (GAIL) - imitation
<https://imitation.readthedocs.io/en/latest/algorithms/gail.html>
- 40 58 CLAUDE_2025-11-10.md
https://github.com/Dutchthenomad/REPLAYER/blob/2f2307c921213f4c4879e444fa938985d0179f6d/deprecated/docs_archive/docs_old/archive/CLAUDE_2025-11-10.md
- 41 What are the most common technical indicators used in deep ...
<https://consensus.app/search/what-are-the-most-common-technical-indicators-used/25PUWgJtQNmDjsXxGEJDjA/>
- 42 [PDF] EQUITY DAY-TRADE AGENT WITH REINFORCEMENT LEARNING
http://stanford.edu/class/msande448/2021/Final_reports/gr2.pdf
- 43 Reinforcement Learning and Stock Trading | Medium - Letian Wang
<https://letian-wang.medium.com/from-reinforcement-gamer-to-reinforcement-trader-8b0a7ef8b53f>
- 44 [PDF] Feature Engineering in Reinforcement Learning for Algorithmic ...
https://repository.tudelft.nl/file/File_f1226238-ebc5-4691-b687-3eb4c5e5663c?preview=1
- 50 Deep Reinforcement Learning in Algorithmic Trading: A Step-by ...
<https://medium.com/funny-ai-quant/deep-reinforcement-learning-in-algorithmic-trading-a-step-by-step-guide-197f39a8be9a>
- 51 Deep Reinforcement Learning in Trading - Quantra by QuantInsti
<https://quantra.quantinsti.com/course/deep-reinforcement-learning-trading>
- 52 A Data Scientist's Approach for Algorithmic Trading Using Deep ...
<https://alpaca.markets/learn/data-scientists-approach-algorithmic-trading-using-deep-reinforcement-learning>

53 Reinforcement Learning for Portfolio Rebalancing - LuxAlgo

<https://www.luxalgo.com/blog/reinforcement-learning-for-portfolio-rebalancing/>

54 How to Evaluate a Trading Strategy Like a Quant | by Yavuz Akbay

<https://medium.com/@yavuzakbay/how-to-evaluate-a-trading-strategy-like-a-quant-fc903e093015>

55 Reinforcement Learning in Trading - QuantInsti Blog

<https://blog.quantinsti.com/reinforcement-learning-trading/>

56 57 59 CLAUDE.md

<https://github.com/Dutchthenomad/VECTRA-PLAYER/blob/340798a0f01393fbbbebdd39e3c65b72f87d0d16a/CLAUDE.md>

60 [PDF] Decision Transformer: Reinforcement Learning via Sequence ...

<https://openreview.net/pdf?id=a7APmM4B9d>

61 Pretrained LLM Adapted with LoRA as a Decision Transformer for ...

<https://arxiv.org/abs/2411.17900>

62 How does reinforcement learning work in financial trading? - Milvus

<https://milvus.io/ai-quick-reference/how-does-reinforcement-learning-work-in-financial-trading>