LLM-Enhanced Self-Evolving Reinforcement Learning for Multi-Step E-Commerce Payment Fraud Risk Detection

Bo Qu[†*], **Zhurong Wang**[†], **Daisuke Yagi**[§], **Zhen Xu**[‡], **Yang Zhao**[†], **Yinan Shan**[†], **Frank Zahradnik**[†]
[†]eBay, [‡]University of Chicago, [§]Etsy
boqu@ebay.com

Abstract

This paper presents a novel approach to e-commerce payment fraud detection by integrating reinforcement learning (RL) with Large Language Models (LLMs). By framing transaction risk as a multi-step Markov Decision Process (MDP), RL optimizes risk detection across multiple payment stages. Crafting effective reward functions, essential for RL model success, typically requires significant human expertise due to the complexity and variability in design. LLMs, with their advanced reasoning and coding capabilities, are well-suited to refine these functions, offering improvements over traditional methods. Our approach leverages LLMs to iteratively enhance reward functions, achieving better fraud detection accuracy and demonstrating zero-shot capability. Experiments with real-world data confirm the effectiveness, robustness, and resilience of our LLM-enhanced RL framework through long-term evaluations, underscoring the potential of LLMs in advancing industrial RL applications.

1 Introduction

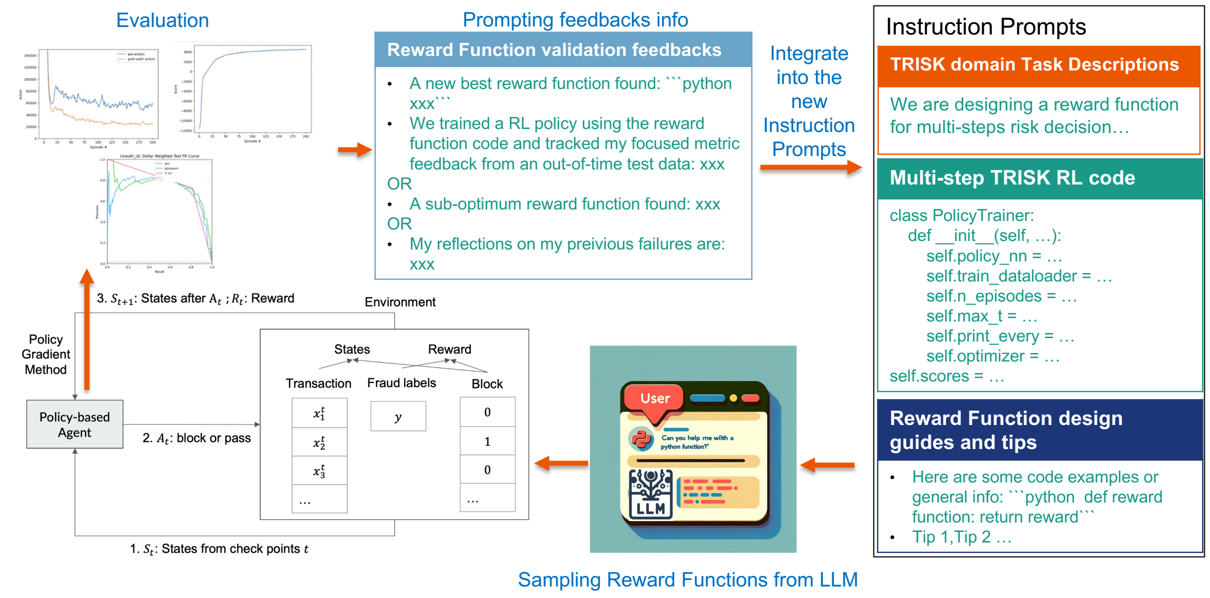
The advancement of LLMs has been remarkable, exemplified by notable developments such as the top-notch model API (OpenAI, 2023) and state-of-the-art open-source models (Dubey et al., 2024) (Jiang et al., 2023) (Jiang et al., 2024) (Team et al., 2024) (Guo et al., 2024). These breakthroughs have propelled LLMs to new heights in various tasks, reaching or even surpassing human capabilities in code generation (Chen et al., 2021), logical reasoning (Kojima et al., 2022), and task planning (Shen et al., 2024). The integration of these advanced capabilities into the domain of e-commerce payment fraud detection presents an exciting frontier for exploration.

Meanwhile, RL has shown its effectiveness in optimizing nondifferential goals and innovating decision strategies in response to environmental changes (Sutton and Barto, 2018) (Russell and Norvig, 2010). Its application in the financial fraud risk domain has seen various approaches, from modeling the sequence of transactions from a single credit card to considering each transaction as a discrete step in a MDP (Mead et al., 2018) (Vimal et al., 2021). Other studies have explored the application of RL in fraud risk alerting systems (Shen and Kurshan, 2020) and discussed its potential without detailed propositions (El Bouchti et al., 2017). While supervised learning (SL) remains prevalent in static fraud detection, it struggles to model sequential dependencies between decision stages and directly optimize business metrics like precision-recall tradeoffs – limitations that RL naturally addresses through reward-driven optimization.

The confluence of LLM's semantic capabilities with RL has sparked interest, particularly in using LLMs as a reward shaper for RL. This innovative approach includes directly feeding the context of the environment to LLMs for action and reward processing (Kwon et al., 2023), using LLMs to define the parameters of the reward function (Yu et al., 2023), or even to design whole rewards function codes (Ma et al., 2023). These efforts have mainly focused on gaming agents and robotic task control, inspiring our exploration into e-commerce payment fraud detection.

E-Commerce payment fraud presents a dynamic challenge necessitating advanced decision-making across three key stages: 1) *Pre-authorization* (Pre-auth) where our platform screens transactions before card issuers' risk assessment, 2) *Issuer check* where card networks validate payment credentials, and 3) *Post-authorization* (Post-auth) where we conduct final risk evaluation after issuer approval. Traditional SL approaches operate isolated classifiers at each stage, failing to model the sequential interdependencies and business constraints (e.g.,

[*]Corresponding Author.



Evaluation
Prompting feedbacks info
Reward Function validation feedbacks
A new best reward function found: ```python xxx```
We trained a RL policy using the reward function code and tracked my focused metric feedback from an out-of-time test data: xxx
OR
A sub-optimum reward function found: xxx
OR
My reflections on my preivious failures are: xxx
Integrate into the new Instruction Prompts
Instruction Prompts
TRISK domain Task Descriptions
We are designing a reward function for multi-steps risk decision…
Multi-step TRISK RL code
class PolicyTrainer:
def __init__(self, …):
self.policy_nn = …
self.train_dataloader = …
self.n_episodes = …
self.max_t = …
self.print_every = …
self.optimizer = …
self.scores = …
Reward Function design guides and tips
Here are some code examples or general info: ```python def reward function: return reward```
Tip 1,Tip 2 …
3. S_{t+1}: States after A_t ; R_t: Reward
Environment
Policy Gradient Method
Policy-based Agent
2. A_t: block or pass
States
Reward
Transaction
Fraud labels
Block
x_1^t
x_2^t
x_3^t
…
y
0
1
0
…
1. S_t: States from check points t
User
Can you help me with a python function?
LLM
Sampling Reward Functions from LLM

Figure 1: The LLM enhanced self-improving RL framework overview. It takes in the task description/instructions, the RL source code, and the example human-designed reward function as the context to generate an executable reward function. We designed an evolutionary algorithm to allow the LLM to evolve the reward function design based on feedback on the performance of the RL agent.

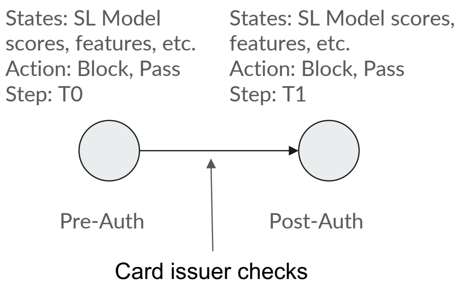


Figure 2: Imagine the buyer transaction risk decision checkpoints pipeline as a Markov Decision Process.

needing to block more potential frauds during Pre-auth to avoid issuer penalties). This fragmentation leads to suboptimal precision-recall balance and excessive manual reviews. RL's strength in constrained sequential optimization makes it uniquely suited to maximize cumulative fraud prevention while respecting stage-specific requirements.

In response, we propose a cutting-edge RL framework that harnesses the power of LLM to autonomously evolve and refine decision-making processes in the payment risk domain, a first in this field. Our contributions are summarized as follows:

LLM-based Reward Function Generation for RL: We introduce a framework using LLMs to autonomously create reward functions that directly optimize precision-recall metrics in the payment risk domain, outperforming human-designed rewards. It uses an evolutionary algorithm for iterative refinement based on RL agent feedback, supporting few-shot/zero-shot creation with/without prior examples. The general process is shown in Figure 1.

Transaction Risk Detection as Constrained MDP: We redefine transaction risk detection as a multistep MDP with stage-specific constraints, solved using policy-based RL like REINFORCE. By integrating transaction stages into a coherent framework (see Figure 2) and aggregating reward signals across stages (detailed in Figure 3), our method outperforms SL's surrogate loss functions through direct optimization of business objectives.

Our research, supported by extensive experiments with real-world e-Commerce transaction data, demonstrates significant improvements in fraud detection performance compared to the existing SL models on our payment system.

2 Methodology

2.1 Designing the MDP and RL Framework

We model the e-commerce transaction process as a finite-horizon MDP, visualized in Figure 2. The system generates state signals from both legacy SL risk model scores and transaction stage indicators (Pre-auth, Post-auth). While there are also many transactional features that can be used as state signals, our experiments primarily use SL scores for state representation due to their proven predictive value leveraging all the features, the framework can theoretically incorporate any transactional features available at each stage. The policy agent uses these state signals to decide between risk responses ("block" or "allow"), with the MDP structure enabling sequential decision-making that supervised learning cannot naturally accommodate.

The agent-environment interaction (Figure 3) defines:

- $\mathcal{S}_i =$ SL scores *and stage indicators* at step i
- $\mathcal{A}_i =$ possible risk responses (block, pass)
- $\mathcal{R}_i = R(\mathcal{S}_i, \mathcal{A}_i)$, the reward function

We maximize the business-driven objective:

$$\begin{aligned} &\text{Maximize} \;\; \$\text{TP} - \$\text{FP} \\ &\text{subject to} \;\; \$\text{TP}_{\text{stage 1}} > \$\text{TP}_{\text{stage 2}} \end{aligned} \tag{1}$$

where dollar-wise \$TP-\$FP optimization directly meets the theoretical goal of our risk business, which corresponds to maximizing fraud prevention while minimizing Loss of the Gross Merchandise Value (GMV) from false positives. The decreasing \$TP constraint reflects practical fraud patterns where early detection captures higher-value fraud attempts.

We employ offline RL with policy gradient methods (REINFORCE (Williams, 1992)) using historical transaction data. To address offline evaluation challenges, we firstly try to train with enough amount of transaction data, and secondly we validate policies on extended test periods (6+ months) demonstrating consistent performance before production deployment.

2.2 Human Reward Function Design

While Equation 1 captures core business objectives, real-world operations require balancing specific precision-recall trade-offs across transaction categories. Here we figured out the reward design that achieve this implicitly through directly considering the optimization constraints instead of the optimization goal itself. By transforming operational constraints into differentiable objectives through algebraic manipulation, we found that it naturally merges into the optimization goal considering the precision block level.

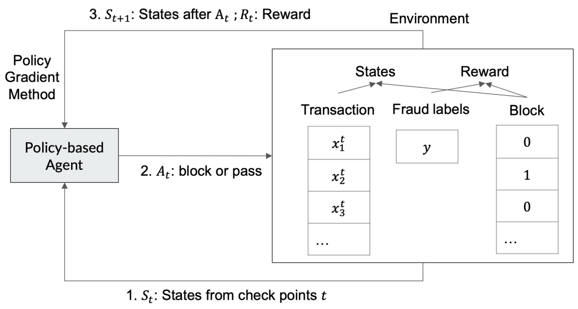


Figure 3: TRISK MDP framework with staged decision points. States incorporate SL risk scores and stage indicators.

Precision Constraint based Reward Function Business requirements ($\$\text{TP}_{\text{stage 1}} > \$\text{TP}_{\text{stage 2}}$) dictate precision thresholds α_i per stage, with $\alpha_1 < \alpha_2$ enforcing stricter precision in later stages. Hence we assume the blocking precision inequality in stage i:

$$\frac{\$\text{TP}_i}{\$\text{TP}_i + \$\text{FP}_i} > \alpha_i \tag{2}$$

we derive the reward function through Lagrangian relaxation:

$$R^i_{\text{precision}}(s, a) = (1 - \alpha_i)\$\text{TP}_i - \alpha_i \$\text{FP}_i > 0 \tag{3}$$

Maximizing this implicitly maximizing (\$TP - \$FP) while maintaining stage-wise constraints by introducing the coefficients in front these terms, derived naturally from the inequality above.

While effective, these human-designed rewards require careful parameter tuning, and in theory there could be more effective designs that need more human efforts to explore. Therefore, we proposed a LLM-enhanced framework automates this exploration by incorporating the specifications of policy performance feedback in natural language, to further enhance the RL reward signals.

2.3 LLM-based Reward Function Optimization

We propose a framework using LLMs to dynamically optimize reward functions in our evolving RL algorithm for e-commerce payment fraud detection.

2.3.1 Algorithm Overview

Our method, detailed in Algorithm 1, employs Enhanced LLM-based Reward Optimization for RL agents, evolving the reward function to boost decision-making. The cycle includes:

1. Initialization with environment $\mathcal{E}$, baseline model $\mathcal{M}_b$, and metrics.

2. Generation of reward candidates by an LLM, guided by temperature for novelty.

3. Validation and use of candidates to train RL agents for fraud detection.

4. Evaluation of detection accuracy and impact, informing reward success.

5. Self-Reflection: top functions update the LLM context; failures refine iterations.

6. Repeat steps 2-5 until iteration or convergence.

To ensure the executability of generated reward functions, we implement a two-step validation process: (1) incorporating basic reward function structure requirements in the prompts, and (2) using preliminary code checks to confirm that generated functions fit the required structure. If a function fails these checks, the LLM regenerates it during the sampling phase, significantly reducing unexecutable cases, without human in the loop.

2.3.2 Customized In-Context Prompt

The initial and iterative instructions provided to the LLM are critical to the success of our algorithm. We construct a domain-specific prompt that outlines the objectives of the reward function, incorporates the RL environment framework, and includes basic requirements and examples. As shown in Figure 1, the prompt is dynamically updated with feedback loop information, allowing the LLM to adapt its generative process to the evolving requirements of the fraud detection task. Examples of prompts are shown in the following boxes, with more detailed content in Appendix A.

Initial Instruction Prompts

You are a reward engineer trying to write reward functions to solve reinforcement learning tasks as effectively as possible. Your goal is to: (1) ... (2) ... The goal of my task is: ..., my codes framework of input data as states and train my policy is shown in the code: "'python {...} "'.
Your reward function should use useful variables from my codes framework as inputs. As some examples, here are some example reward functions proposed by humans: "'python {...} "', and here is the best reward function signature so far: "'python {...} "' ... The output of the reward function should consist of: (1) ... (2) ... ...

Feedback Prompts

We trained a RL...:
1. RL Agent Training info: ...
2. Test evaluation info: ...
Moreover, the ratio between the bad GMV blocked by first step and the bad GMV blocked by second step is: {...}/{...} ...
Error occurred during training: {...}
Error occurred during evaluating: {...}

2.3.3 Zero-shot and Few-shots setups

Our approach supports both zero-shot and few-shot capabilities. In the zero-shot setup, the algorithm generates reward functions based on general component descriptions rather than predefined human-designed functions. For the few-shot setup, detailed examples of human-crafted reward functions are included in the prompt, allowing the model to reference specific code and build on these exemplars.

Feedback and success metrics play a crucial role in optimizing the reward function, especially in zero-shot scenarios. Feedback comprises policy evaluation results, such as precision-recall on test data, error reports, and comparative evaluations of previous best and sub-optimal rewards. Importantly, in cases where no sub-optimal reward is found, a reflection process allows the LLM to summarize insights from failed reward functions, integrating this experience into instructions for subsequent iterations, as described in line 26 of Algorithm 1. This reflective feedback is vital for zero-shot cases.

Algorithm 1 LLM-based Reward Function Optimization for RL Agent

```
Require: $N_{iter}, N_{samples}, N_{episodes}, \theta_{recall}, R_{scores}$
1: Initialize environment $\mathcal{E}$, baseline model $\mathcal{M}_b$, and evaluation parameters
2: $f_{best} \leftarrow$ InitializeBestRewardFunction(), Initialize LLM temperature parameters
3: Load baseline model performance and set evaluation criteria
4: for $iter = 1$ to $N_{iter}$ do
5:     Initialize feedback and success lists: $feedbacks, success$
6:     Update LLM temperature based on feedback loop criteria
7:     for $sample\_i = 1$ to $N_{samples}$ do
8:         Sample and validate $f_{reward}^{sample\_i}$ using LLM with temperature control
9:         if valid $f_{reward}^{sample\_i}$ then
10:            Save $f_{reward}^{sample\_i}$, proceed to training
11:        else
12:            Re-sample $f_{reward}^{sample\_i}$
13:        end if
14:    end for
15:    for each valid $f_{reward}^{sample\_i}$ do
16:        $\mathcal{A}_i \leftarrow$ TrainAgent($\mathcal{E}, f_{reward}^{sample\_i}, N_{episodes}$)
17:        $feedback_i, success_i \leftarrow$
           EvaluateAgent($\mathcal{A}_i, \mathcal{M}_b, \theta_{recall}, R_{scores}$)
18:        Append $feedback_i$ to $feedbacks$ and $success_i$ to $success$
19:    end for
20:    Update $f_{best}$ based on evaluation results, Update LLM temperature and instructions for next iteration based on feedback loop outcomes
21:    if new $f_{best}$ found then
22:        Update system instructions for LLM to include new best reward function details
23:    else if sub-optimal reward function found then
24:        Update system instructions for LLM to include sub-optimal reward function details as feedback
25:    else
26:        Let LLM summarize reflections based on the failed reward functions info and include its experience into the instructions for next iteration
27:    end if
28: end for
```

2.3.4 Interpretability of LLM-Generated Reward Functions

While the proposed framework leverages LLMs to automatically evolve reward functions for RL agents, it is important to acknowledge that such LLM-generated reward functions inherently carry a degree of "black-box" behavior, especially in zero-shot settings. To enhance interpretability, we embed domain-specific contextual information into the prompts provided to the LLM.

In both zero-shot and few-shot reward function design prompts, we explicitly define domain-specific contexts such as key business metrics — $TP, $FP, $TN, and $FN — along with their implications in fraud detection (lines 6–9 in the prompt example below). These definitions are paired with optimization objectives and constraints within the domain context (lines 10–11), further reinforced by additional descriptions in the instruction prompts and feedback mechanisms detailed in Section 2.3.2. This structured context guides the LLM to generate reward functions that align closely with real-world business requirements. Take the zero-shot reward design as an example: in Listing 1, the LLM incorporates terms such as $FP and $FN, indicating its understanding of the trade-offs between $TP vs. $FP and $TN vs. $FN. It also assigns higher weights to early-stage rewards (e.g., reward *= 1.2 at current_step == 0 and reward *= 0.9 at current_step == 1), reflecting the business requirement that detecting fraud earlier yields greater value.

Domain-Specific Context Prompts for Reward Function Design

1. element in action either equals 0 or 1;
2. action == 1 means the transactions that were taken blocking action, action == 0 means the transactions that were taken pass action;
3. element in target either equals 0 or 1;
4. target == 1 means the transactions that are tagged as fraud risk, target == 0 means the transactions are not tagged as risk;
5. wgt is the tensor of dollarwise weight for each transaction;
6. e.g. ((action==1) & (target==1) * wgt) means the tensor that have the True Postive GMV value where (action==1) & (target==1);
7. e.g. ((action==1) & (target==0) * wgt) means the tensor that have the False Positive GMV value where (action==1) & (target==0);
8. e.g. ((action==0) & (target==0) * wgt) means the tensor that have the True Negative GMV value where (action==0) & (target==0);
9. e.g. ((action==0) & (target==1) * wgt) means the tensor that have the False Negative GMV value where (action==0) & (target==1);
10. the general goal of this reward function is to drive the agent to increase True Postive GMV and True Negative GMV, decrease False Positive GMV and False Negative GMV;
11. this reward function need to drive the agent to block more potential True Postive GMV at the current_step == 0 than at the current_step == 1.

Despite these efforts, certain aspects — such as why specific parameter choices lead to particular precision-recall outcomes on certain test data—remain difficult to fully interpret. Therefore, we complement the validation of the reward function design with long-term evaluations (Test L in Table 1), demonstrating the stability and practical effectiveness of the evolved reward functions over time.

```python
def get_reward(current_step, action, target, wgt):
    reward = (action * target * wgt).float()
    if current_step == 0:
        reward *= 1.2
    elif current_step == 1:
        reward *= 0.9
    fn = ((1 - action) * target * wgt).float()
    reward -= fn * 0.5
    fp = ((action * (1 - target) * wgt).float())
    reward -= fp * 0.1
    low_weight_penalty = (action * (wgt < 50)).float()
    reward -= low_weight_penalty * 0.005
    reward /= wgt
    return reward
```

Listing (1) Original zero-shot reward function design by Mixtral8X7B. The calculation of rewards and penalties in both steps is uniquely different compared to Equation 3 above.

```python
def get_reward(current_step, action, target, wgt):
    gamma_positive = 1.15
    gamma_negative = 0.9
    alpha = 1.2
    reward = 0
    if current_step == 0:
        reward = gamma_positive * (
            ((action == 1) & (target == 1)) * wgt -
            ((action == 1) & (target == 0)) * (alpha * 0.005) * wgt -
            0.15 * ((action == 0) & (target == 1)) * wgt
        )
    elif current_step == 1:
        reward = gamma_negative * (
            ((action == 1) & (target == 1)) * wgt -
            ((action == 1) & (target == 0)) * (alpha * 0.002) * wgt -
            0.10 * ((action == 0) & (target == 1)) * wgt
        )
    return reward
```

Listing (2) Original few-shot reward function design by Mixtral8X7B. This design introduces unique reward terms compared to Equation 3 above, rather than simply adjusting the parameters of the human-designed version.

Figure 4: Reward function designs evolved by Mixtral8X7B in different contexts: Listing (1) Zero-shot context, Listing (2) Few-shot context.

Table 1: Experiment Datasets.

Dataset	Time Window	Total	Fraud Label
Train	2023-09-01 to 2023-09-14	2,136,590	28,226
Test S	2023-09-15 to 2023-09-30	522,105	825
Test L	2023-11-01 to 2024-04-30	6,174,069	7,834

Table 2: Performance of Policy Agent vs. Baseline, on Test S.

Recall Levels	Baseline $Prec	RL Agent $Prec	Bad GMV Catch Ratio
@80%	66.57%	**69.65%**	**9.79**
@85%	58.79%	**64.22%**	**15.32**
@90%	51.27%	**55.7%**	**13.36**

2.3.5 Generalizability Discussion

State-of-the-art approaches, such as those presented by (Ma et al., 2023), have employed evolutionary loops to demonstrate the robustness of these methods in optimizing RL training processes within different robotics tasks. However, these frameworks are primarily tailored to the specific data and scenarios encountered in robotics, limiting their direct applicability to our domain. Therefore, our work introduces this novel adaptation of evolutionary loops for tasks in e-commerce risk detection, for the first time. By doing so, we first demonstrate that this evolutionary reward design loop, leveraging LLMs, can be effectively generalized to e-commerce payment fraud scenarios. Theoretically, this approach can also be extended to other RL tasks within this domain that share similar data structures and objectives.

3 Experiments

3.1 Datasets and Evaluation Metrics

We used real-world transaction data focusing on Pre-auth and Post-auth stages. SL models (gradient boost machines) scores $\mathcal{S}_i = \{Scr_{i0}, \cdots, Scr_{ij}\}$ on the 2 stages, and stage indicators, represented the RL state. Data were split, labeled with our key fraud signals, and evaluated on out-of-time test sets. Table 1 shows dataset details. Test S, with 522K transactions, allows for quick performance comparisons but may introduce more variance due to its size. In contrast, Test L, with 6.17M transactions, offers more robust validation.

We assess performance using a metric for dollar-wise precision ($Precision) at key dollar-wise recall ($Recall) levels, calculated by our main fraud label. This metric is crucial as it aims to maximize legitimate GMV by minimizing $FP transaction values at a given risk level. For the RL agent scores, we find combinations of blocking score thresholds across two stages to achieve the desired overall $Recall, then observe the $Precision. For the baseline model, we use the Pre-auth SL model score, which is most commonly employed by the policy, to observe this metric. Due to the complexity of human analysis in business practice, no cross-stage policy has been designed previously using SL model scores as a baseline. Which is also why we need to propose our RL solution in the first place.

3.2 Experimental Results and Analysis

Part 1: Human-designed Reward Function: In the first segment, a single RL agent was trained using a 3-layer neural network with dimensions [8, 32, 8], incorporating dropout layers and GELU activation functions. The model processed a four-dimensional input consisting of representative scores from legacy SL models, which served as the state representation. The output was the probability of taking the "block" action. Training was conducted using the REINFORCE algorithm with the Adam optimizer.

Multiple trials stabilized results, Table 2 shows enhanced performance and risk detection efficiency, with the agent blocking more fraudulent GMV in the Pre-auth stage.

All training in part 1 was performed on a machine equipped with a single V100 GPU (32GB VRAM), 32 CPU cores, and 450GB of RAM. With our current implementation, iterating over 200 training epochs — generally sufficient for observing convergence in our experiments — took approximately 20 minutes per epoch. Each iteration involved processing the full training dataset, as detailed in Table 1.

Part 2: LLM-enhanced Reward Function: We employed LLM-enhanced rewards using models like Mixtral-8x7B, LLaMa-3-8B, and Gemma7B. Experiments included zero-shot and few-shot setups with varying LLM prompts. Algorithm 1 parameters included $N_{iter} \approx 60$, $N_{samples} \approx 10$, $N_{episodes} \approx 150$, and $\theta_{recall} \in [80\%, 85\%, 90\%]$. Results are in Table 3.

Zero-shot scenarios used descriptive prompts without reward function examples, leading to competitive reward designs, as shown in Listing (1). Few-shot scenarios also allowed LLMs to mod-

Table 3: Zero-shot and Few-shot Performance Comparison of LLMs in LLM+RL Approach, on Test S.

Recall Levels	Baseline $Prec	Zero-shot Evolved RL agent $Prec			Few-shot Evolved RL agent $Prec		
		Mixtral-8x7B	Gemma7B	LLaMa-3-8B	Mixtral-8x7B	Gemma7B	LLaMa-3-8B
@80%	66.57%	**72.71%**	**73.27%**	**72.86%**	**73.41%**	**73.53%**	**73.74%**
@85%	58.79%	**69.62%**	**65.42%**	**69.40%**	**70.73%**	**69.87%**	**71.70%**
@90%	51.27%	**57.42%**	**53.65%**	**57.06%**	**58.00%**	**56.93%**	**55.90%**

ify and create reward functions, as shown in Listing (2), improving performance metrics. Zero-shot setups required more iterations, indicating optimization potential, but overall, LLM-enhanced approaches showed adaptability and innovation.

Each complete training iteration, encompassing LLM inference, RL agent training, and performance evaluation, required approximately 40 minutes. All experiments in part 2 were conducted on a machine equipped with 2 V100 GPUs (32GB VRAM), 32 CPU cores, and 450GB of RAM, with LLMs loaded in 4-bit precision ($load\_in\_4bit = True$) to reduce VRAM consumption. The primary computational bottlenecks were identified as LLM inference and policy evaluation. These components represent key areas for future optimization in the implementation pipeline.

Part 3: Long-term Evaluation: To test RL agent robustness over time, we extended evaluation on Test L covering six additional months. Using the same RL agent, we analyzed performance with $Prec metric against a baseline model at similar $Recall thresholds for all LLMs in both zero-shot and few-shot scenarios.

Figure 5 shows RL agents consistently outperforming the baseline over time. Figure 6 illustrates zero-shot scenarios where RL agents maintained superior performance.

These evaluations highlight our LLM-enhanced RL framework's durability and effectiveness in real-world applications, supporting continuous deployment without frequent retraining. More results are in Appendix B.

3.3 Production Efficiency

Due to the compact architecture and lightweight design of the RL agent network described above, the model supports efficient deployment across both transaction stages. In production, it achieves inference latencies of less than 50 milliseconds using standard CPU infrastructure, making it suitable for real-time fraud detection at scale.

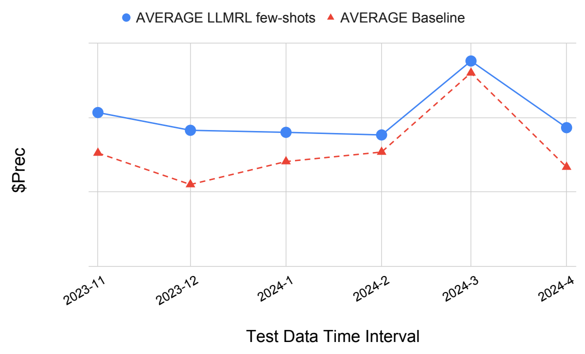


Figure 5: Averaged blocking $Prec@$Recall from 3 LLM guided RL agents, in the few-shots scenario, on Test L.

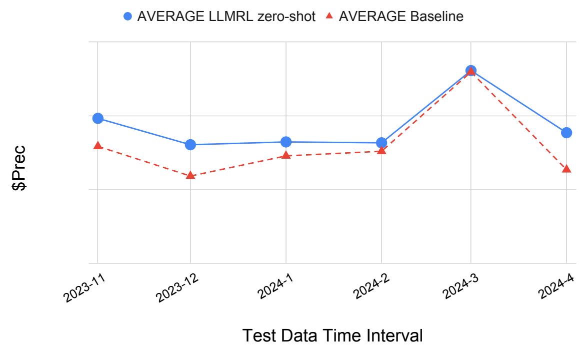


Figure 6: Averaged blocking $Prec@$Recall from 3 LLM guided RL agents, in the zero-shots scenario, on Test L.

4 Conclusion

This study introduces an RL and LLM integration framework for e-Commerce fraud detection, conceptualizing risk assessment as an MDP and enabling dynamic sequential strategies. Our approach, using LLMs to refine reward functions, surpasses traditional human-designed functions in efficiency and zero-shot capability. Empirical tests confirm its superiority over our conventional SL model, with six-month evaluations demonstrating robust performance. The lightweight architecture, is practical for industrial adoption. Future work includes generalizing to more sequential scenarios of risk prevention, and exploring online RL.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaoqing Ellen Tan, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aaron Grattafiori, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alex Vaughan, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Franco, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, Danny Wyatt, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Firat Ozgenel, Francesco Caggioni, Francisco Guzmán, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Govind Thattai, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Karthik Prasad, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kun Huang, Kunal Chawla, Kushal Lakhotia, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian

Khabsa, Manav Avalani, Manish Bhatt, Maria Tsimpoukelli, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikolay Pavlovich Laptev, Ning Dong, Ning Zhang, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Rohan Maheswari, Russ Howes, Ruty Rinott, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Kohler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vítor Albiero, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaofang Wang, Xiaojian Wu, Xiaolan Wang, Xide Xia, Xilun Wu, Xinbo Gao, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yuchen Hao, Yundi Qian, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, and Zhiwei Zhao. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Abdelali El Bouchti, Ahmed Chakroun, Hassan Abbar, and Chafik Okar. 2017. Fraud detection in banking using deep reinforcement learning. In *2017 Seventh International Conference on Innovative Computing Technology (INTECH)*, pages 58–63. IEEE.

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

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. 2023. Reward design with language models. *arXiv preprint arXiv:2303.00001*.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*.

Adrian Mead, Tyler Lewris, Sai Prasanth, Stephen Adams, Peter Alonzi, and Peter Beling. 2018. Detecting fraud in adversarial environments: A reinforcement learning approach. In *2018 Systems and Information Engineering Design Symposium (SIEDS)*, pages 118–122. IEEE.

R OpenAI. 2023. Gpt-4 technical report. arxiv 2303.08774. *View in Article*, 2:13.

Stuart J Russell and Peter Norvig. 2010. *Artificial intelligence a modern approach*. London.

Hongda Shen and Eren Kurshan. 2020. Deep q-network-based adaptive alert threshold selection policy for payment fraud systems in retail banking. In *Proceedings of the First ACM International Conference on AI in Finance*, pages 1–7.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Siddharth Vimal, Kanishka Kayathwal, Hardik Wadhwa, and Gaurav Dhama. 2021. Application of deep reinforcement learning to payment fraud. *arXiv preprint arXiv:2112.04236*.

Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256.

Wenhao Yu, Nimrod Gileadi, Chuyuan Fu, Sean Kirmani, Kuang-Huei Lee, Montse Gonzalez Arenas, Hao-Tien Lewis Chiang, Tom Erez, Leonard Hasenclever, Jan Humplik, et al. 2023. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*.

A Prompts Design for the LLM RL framework

In this section, we provide the prompts of our LLM RL framework.

Prompt 1: Initial Instruction Prompts

You are a reward engineer trying to write reward functions to solve reinforcement learning tasks as effective as possible. Your goal is to: (1) write a reward function for the environment that will help the agent learn the task described below. (2) try to write improved or try different parameters in the reward function comparing to the reward functions found so far, based on analyzing the provided reward function feedback information below. The goal of my task is: Design a reward function that enables the RL agent to make more effective decisions across 2 steps for improved overall performance in identifying and blocking risky transactions comparing to a baseline scores in the 1st step, my codes framework of input data as states and train my policy is shown in the code: “‘python {...} “‘.

Prompt 2: Code Generation Instruction Prompts

Your reward function should use useful variables from my codes framework as inputs. As some examples, here are some examples reward functions proposed by human: “‘python {...} “‘, and here is the best reward function signature so far: “‘python {...} “‘ Since the reward function will be decorated with @torch.jit.script, please make sure that the code is compatible with TorchScript (e.g., use torch tensor instead of numpy array).
Make sure any new tensor or variable you introduce is on the same device as the input tensors. The output of the reward function should consist:
(1) the completed reward function.
(2) the reward code’s input attributes must follow the format:"def get_reward(current_step,action,target,wgt):".
(3) the code output should be formatted as a python code string: "“‘python ... “‘".
(4) if you have extra functions defined in the reward function, also output these functions completely in one code block.
(5) your codes and the related annotations must be consistent.
(6) it is encouraged to only output your completed reward function python codes in the beginning of your outputs, for the ease of code extraction.
(7) remember to use the backslash properly as a line continuation where you separate one logic line into multiple physical lines for better readability.

Prompt 3: Additional Reward Generation Instruction Prompts with Domain-Specific Context

information of the get_reward:
def get_reward(current_step,action,target,wgt):
# current_step is one integer;
# if the agent is in step 0, then current_step == 0;
# if the agent is in step 1, then current_step == 1;
# current_step either equals 0 or 1 in get_reward function;
# action and target and wgt are tensors in size (transaction_batch_size,);
# element in action either equals 0 or 1;
# action == 1 means the transactions that were taken blocking action, action == 0 means the transactions that were taken pass action;
# element in target either equals 0 or 1;
# target == 1 means the transactions that are tagged as fraud risk, target == 0 means the transactions are not tagged as risk;
# wgt is the tensor of dollarwise weight for each transaction.;
# e.g. ((action==1) & (target==1) * wgt) means the tensor that have the True Postive GMV value where (action==1) & (target==1);

# e.g. ((action==1) & (target==0) * wgt) means the tensor that have the False Positive GMV value where (action==1) & (target==0);
# e.g. ((action==0) & (target==0) * wgt) means the tensor that have the True Negative GMV value where (action==0) & (target==0);
# e.g. ((action==0) & (target==1) * wgt) means the tensor that have the False Negative GMV value where (action==0) & (target==1);
# the general goal of this reward function is to drive the agent to increase True Postive GMV and True Negative GMV, decrease False Positive GMV and False Negative GMV;
# this reward function need to drive the agent to block more potential True Postive GMV at the current_step == 0 than at the current_step == 1;
# the returned reward also need to be a tensor in size (transaction_batch_size,) or (transaction_batch_size,1) , it will be aggregated outside this get_reward function
return reward

Prompt 4: Feedback Prompts

We trained a RL policy using the new found reward function code and tracked my focused metric feedback from a out-of-date test data:
1. RL Agent Training info: after training in {...} episodes, the final blocking action number of the RL agent in first step is: {...}, and the final blocking action number of second step is: {...}, and the final reward value is: {...} comparing to the initial reward value is: {...}. Normally we hope to observe the RL agent take more blocking action in the first step than in the second step, and the final reward value should be larger than the initial value.
2. Test evaluation info: after 2 steps actions of a policy agent, we observed the final best precision performance by the agent under some targeting recall thresholds levels: {...} and compare with the baseline model, the goal is have better precision compare to the baseline model. The detail of the observations are: Our 2 steps policy agent can reach the similar recall:{...} and the agent can reach at best the precision: {...}. Moreover, the ratio between the bad GMV blocked by first step and the bad GMV blocked by second step is: {...}/{...}, and the ratio between the total GMV blocked by first step and the total GMV blocked by second step is {...}/{...};
Error occurred during training: {...}
Error occurred during evaluating: {...}

Prompt 5: Reflection Prompts if No Usable Reward Function Found

However, after an iteration of reward designs and validations, all of your designed reward functions failed in either training or evaluation, your designs and their regarding failure info are listed here: {...}
With all the feedback information, reflect the failed experience regards to your reward functions and output a detailed guidance of reward function design for yourself briefly, in less than length of 1000 tokens:

Prompt 6: Reflection Prompts if A Better Reward Function Found

The previous best reward function's policy agent performance: when the recall threshold is {...}, the baseline model can reach the precision: {...}. A better new found reward function in iteration {...}:{...}.

Prompt 7: Reflection Prompts if A Sub-optimal Reward Function Found

You found a sub-optimal new reward function in iteration {...}:{...}, which has worse performance than the previously best reward function.

B Long-term Test evaluations with different LLM

In this appendix, we present detailed figures illustrating the performance of different models evaluated in this study.

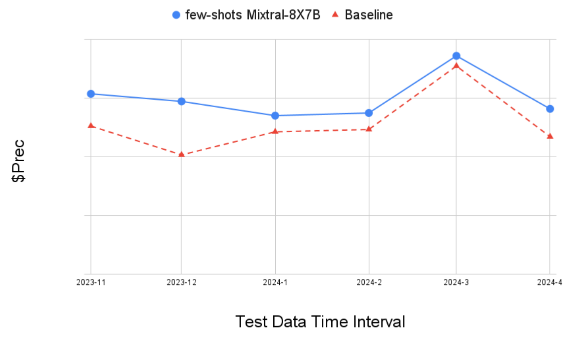


Figure B.1: Averaged blocking $Prec@$Recall from Mixtral-8X7B guided RL agents, in the few-shots scenario.

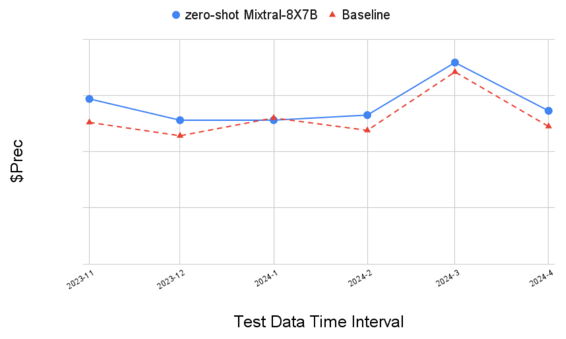


Figure B.4: Averaged blocking $Prec@$Recall from Mixtral-8X7B guided RL agents, in the zero-shot scenario.

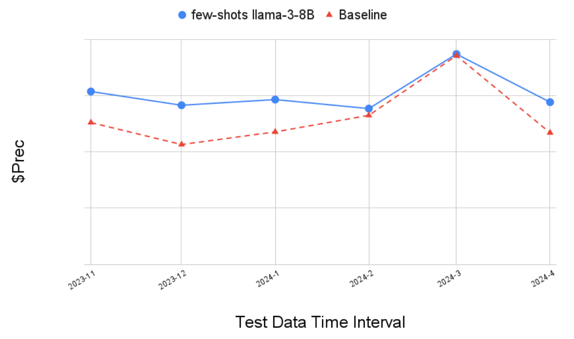


Figure B.2: Averaged blocking $Prec@$Recall from LLaMa-3-8B guided RL agents, in the few-shots scenario.

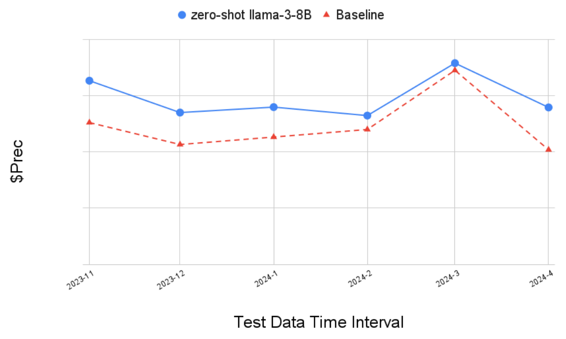


Figure B.5: Averaged blocking $Prec@$Recall from LLaMa-3-8B guided RL agents, in the zero-shot scenario.

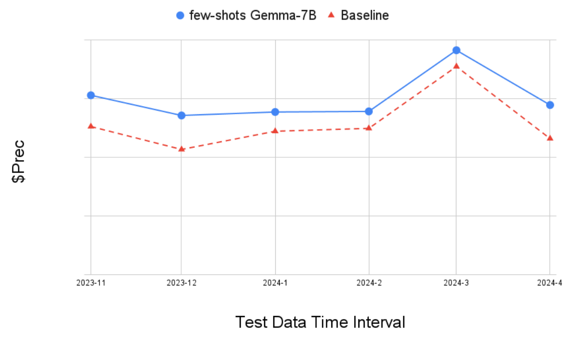


Figure B.3: Averaged blocking $Prec@$Recall from Gemma7B guided RL agents, in the few-shots scenario.

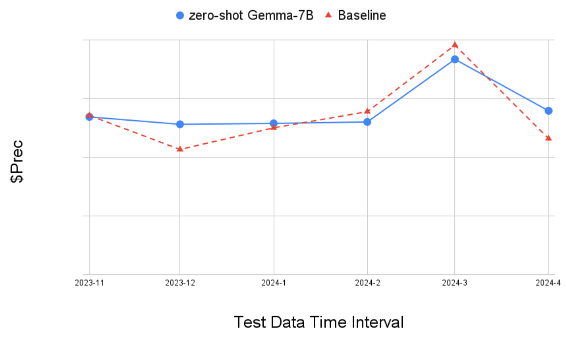


Figure B.6: Averaged blocking $Prec@$Recall from Gemma7B guided RL agents, in the zero-shot scenario.