

# TalkToAgent: A HUMAN-CENTRIC EXPLANATION OF REINFORCEMENT LEARNING AGENTS WITH LARGE LANGUAGE MODELS

**Anonymous authors**

Paper under double-blind review

## ABSTRACT

Explainable Reinforcement Learning (XRL) has emerged as a promising approach in improving the transparency of Reinforcement Learning (RL) agents. However, there remains a gap between complex RL policies and domain experts, due to the limited comprehensibility of XRL results and isolated coverage of current XRL approaches that leave users uncertain about which tools to employ. To address these challenges, we introduce TalkToAgent, a multi-agent Large Language Models (LLM) framework that delivers interactive, natural language explanations for RL policies. The architecture with five specialized LLM agents—*Coordinator*, *Explainer*, *Coder*, *Evaluator*, and *Debugger*—enables TalkToAgent to automatically map user queries to relevant XRL tools and clarify an agent’s actions in terms of either key state variables, expected outcomes, or contrastive explanations. Moreover, our approach extends previous contrastive explanations by deriving alternative scenarios from qualitative behavioral descriptions, or even new rule-based policies. We validated TalkToAgent on quadruple-tank process control problem, a well-known nonlinear control benchmark. Results demonstrated that TalkToAgent successfully mapped user queries into XRL tasks with high accuracy, and coder–debugger interactions minimized failures in new policy generation. Furthermore, qualitative evaluation confirmed that TalkToAgent effectively interpreted agent’s actions and contextualized their meaning within the problem domain.

## 1 INTRODUCTION

Deep reinforcement learning (DRL) has significantly improved the scalability of RL algorithms by leveraging deep neural networks to handle high-dimensional state and action spaces. This advancement has enabled broader adoption of RL, including in safety-critical industrial applications such as process control or energy systems (Park et al., 2025; Perera & Kamalaruban, 2021). Despite these advancements, the inherent black-box nature of DRL models makes their decision-making difficult to interpret. This is particularly critical in safety-sensitive domains, where trust and transparency are essential. While explainable reinforcement learning (XRL) has demonstrated promise in interpreting the agent’s behavior (Milani et al., 2024; Puiutta & Veith, 2020), there still exist critical challenges that hinder the direct adoption of existing XRL methods in industrial applications.

One major challenge in current XRL approaches is that most methods offer only a limited view of the agent’s behavior. Since RL agents make sequential decisions aimed at maximizing future rewards, a wider range of factors influence the agent’s actions compared to supervised learning settings. For example, an action may depend not only on the current state, but also on its long-term consequences. To capture this broader perspective, an effective XRL framework should flexibly support diverse explanation types aligned with user intent.

In addition, the lack of familiarity with XRL techniques makes it challenging for non-experts to effectively utilize them. Users may struggle to identify which XRL method is most appropriate for addressing their question. Even when relevant XRL outputs are provided, deriving meaningful interpretations from them is still nontrivial. This highlights the need for a natural language interface

Previous studies	FI	EO	CE-A	CE-B	CE-P	NL
Xi-Jia et al. (2025)	✓	✗	✗	✗	✗	✓
Amitai et al. (2024)	✗	✓	✓	✗	✗	✗
Towers et al. (2024)	✓	✓	✓	✗	✗	✗
Our work	✓	✓	✓	✓	✓	✓

Table 1: Comparison of supported explanation types by our method and previous studies. (FI: Feature Importance, EO: Expected Outcomes, CE-A: Action-based Contrasts, CE-B: Behavior-based Contrasts, CE-P: Policy-based Contrasts, NL: Natural Language)

that can both map user intent to appropriate XRL strategies and deliver results in a comprehensible form.

Lastly, contrastive explanations, which explain an agent’s decisions by contrasting them with alternative ones, are typically limited to simple action substitutions in existing XRL studies. This hinders their practical applications where users often express intent through abstract or qualitative behavioral descriptions. For example, when users request alternative scenarios where agent’s behaviors are described in natural language (e.g., *conservative* or *opposite* behavior), or when they want to compare the agent with alternative policies, existing methods fail to accommodate such cases. Addressing these needs would significantly broaden the scope of explainability in practice.

To address these challenges, we propose *TalkToAgent*, a multi-agent Large Language Model (LLM) framework that interprets RL policies through natural language interactions. *TalkToAgent* is designed to handle diverse user queries, including questions about feature importance, expected outcome, and contrastive scenarios, by mapping them to relevant XRL tools. The output from *TalkToAgent* includes figures generated from XRL tools and natural language interpretations generated by LLM agents, helping the user understand the behavior of the agent in multimodal fashion.

Another key novelty of *TalkToAgent* lies in its versatile generation of contrastive scenarios. Beyond the action-level interventions, *TalkToAgent* supports comparison of diverse alternative strategies by translating high-level user requests into action trajectories or executable policy code. Our main contributions are summarized below:

- We introduce *TalkToAgent*, a multi-agent LLM-based XRL explainer framework that answers natural language queries about the intentions behind an agent’s actions.
- *TalkToAgent* unifies solutions to various XRL tasks within a single framework, achieving a mapping accuracy of 97.5% with the GPT-4.1 model.
- We expand contrastive explanations beyond single-step actions to include qualitative behavioral terms and rule-based policies, while attaining minimal failure rate through interactions among coder, evaluator, and debugger agents.
- *TalkToAgent* improves the interpretability of RL agents by providing domain-aware textual explanations of XRL visualizations.

## 2 RELATED WORK

**Explainable Reinforcement Learning (XRL)** XRL has emerged as a pivotal tool for enhancing the transparency and trustworthiness of RL agents, particularly in safety-sensitive domains such as autonomous driving (Atakishiyev et al., 2024) and process control (Szatmári et al., 2024). Recent XRL literature can be broadly categorized into four major directions: *feature importance*, *expected outcome*, *contrastive explanations*, and *policy-level* explanations.

*Feature importance* methods elucidate which state features or pixels most influence the agent’s decisions. Traditional explainable AI techniques such as saliency maps (Wang et al., 2019; Atrey et al., 2019), interpretable surrogate models (Liu et al., 2018), and post-hoc attributions (Zhang et al., 2021) have been adopted to identify critical features.

Agent’s *expected outcome* helps identify when and where the agent expects important events and rewards to occur from executing a certain action. It can be attained by decomposing Q-values into

temporal dimension (Ruggeri et al., 2025; Towers et al., 2024), or extracting future trajectories via belief maps and forward simulations (Yau et al., 2020; van der Waa et al., 2018).

A *contrastive explanation* illustrates what would have happened if the agent had behaved differently, helping users better understand the rationale behind its actual decision (Miller, 2019). Prior works have integrated such contrastive explanations with other explainable components, including future trajectories (Amitai et al., 2024), causal graphs (Madumal et al., 2020), and decomposed rewards (Lin et al., 2020). These approaches typically intervene on action values within a single timestep, limiting their expressiveness in the explanations.

Lastly, *policy-level* explanations offer a global perspective on the agent’s behavior across the entire state space. These approaches include identifying regions where the agent behaves suboptimally or exhibits high uncertainty (Sequeira & Gervasio, 2020; Huang et al., 2018), or analyzing the dynamics between physically meaningful clusters of states (Zahavy et al., 2016).

Such diverse nature of XRL techniques highlights the need for systems that capture the intent behind user queries and automatically map them to the most appropriate XRL techniques. In addition, extending contrastive explanations beyond simple action interventions to include high-level behavioral terms or alternative policies would enrich their applicability.

**Natural Language-based Explainers** Natural language has been gaining increasing attention in the field of explainable artificial intelligence as it provides an interactive and accessible medium for users. In supervised settings, Slack et al. (2023) mapped user intent to relevant explanation types such as feature importance, counterfactuals, and feature interactions. In RL, Hayes & Shah (2017) proposed a template-based system to answer natural language inquiries about robot behavior. Xi-Jia et al. (2025) leveraged LLMs to explain local decision paths of states influencing an agent’s actions.

Building on these efforts, this work maps natural language queries directly to multiple XRL tools, and provides textual interpretations along with the resulting visualizations.

**Relations to previous works** We compare the types of explanations supported by our proposed TalkToAgent framework with those of related work in Table 1. The explanation types range from feature importance (FI) and expected outcomes (EO) to various forms of contrastive explanations (CE-A, CE-B, CE-P). Detailed descriptions of each explanation type are provided in the Methods section.

To the best of our knowledge, the only prior work that addressed natural language-based XRL queries is Xi-Jia et al. (2025), which focused solely on *feature importance* explanations. Although Amitai et al. (2024) and Towers et al. (2024) explored contrastive explanations by comparing future rewards, their methods are limited to single-step interventions and lack adaptability to more complex scenarios.

## 3 METHODS

### 3.1 OVERVIEW

The overall structure of *TalkToAgent*, illustrated in Figure 1, consists of five specialized LLM agents with key Python functions that implement XRL techniques. The workflow begins when a user poses a natural language query about a trained RL agent. A set of agents collaborate to interpret the query, invoke appropriate XRL tools, and return multimodal explanations. The system also features automated code generation for new reward functions or policies, enabling flexible responses to diverse user queries. Details about each agent’s specific role and their interactions are described in subsequent sections.

### 3.2 MULTI-AGENT LLM STRUCTURE

We design a multi-agent framework consisting of five specialized LLM agents, where each is assigned a distinct role and corresponding instruction in the explanation process. Through interactions among these agents, the system is able to automate the explanation pipeline.

162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215

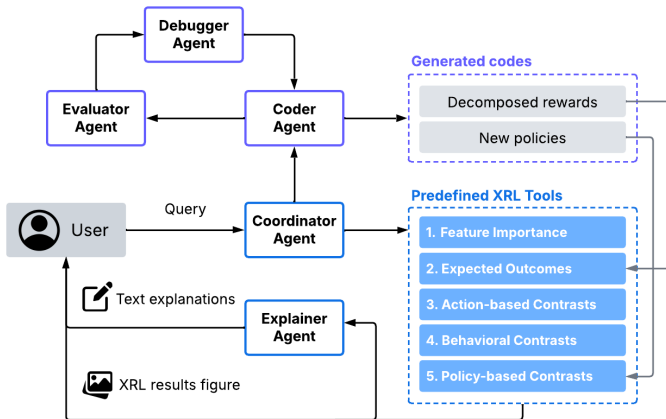


Figure 1: Overall framework of *TalkToAgent*.

- The **coordinator** interprets user queries and determines the appropriate XRL tools along with its relevant arguments. If a query requested an operation outside the available XRL function scope, the coordinator would explicitly return an error.
- The **explainer** provides natural language explanations for the generated XRL output figures. Given domain knowledge prior to the explanation, it is further instructed to relate the interpretations to the context of the current system, offering implications for domain experts.
- The **coder** constructs Python code when requested by the XRL function tools. In our work, the coder agent is demanded to generate two types of codes: decomposed reward functions and alternative rule-based policies, based on instructions from the coordinator. To reduce potential errors, it is instructed to produce a code that closely follows the structure of the original reward or policy code. The generated code serves as the user-defined rewards or policy component for assisting explanations.
- The **evaluator** validates whether the code generated by the coder agent correctly implements the requested reward or policy. It prevents against generating hallucinated policies by detecting deviations from the intended logic and raising exceptions when necessary.
- Whenever execution errors or hallucinations are detected by the evaluator, the **debugger** analyzes both the code and the resulting traceback to generate debugging instructions. The instructions are then sent back to the coder for iterative refinement, ensuring that the final policy is both executable and aligned with the user’s intent.

All agents are implemented using OpenAI’s GPT-4.1 model by default, with different GPT variants used for sensitivity analysis. System descriptions and environment parameters are provided as contextual prompts to ensure domain-aware reasoning. Full prompts for each agent are available in the supplementary materials.

### 3.3 AUTOMATED CODE GENERATION

The code generation procedure using the agents defined previously is outlined in Algorithm 1. If the generated code fails during execution or the evaluator agent determines that the resulting trajectory does not align with the user’s intent, an error is raised. This triggers the debugger agent to produce debugging guidance, prompting the coder agent to refine the policy accordingly.

### 3.4 REINFORCEMENT LEARNING SETUP

The RL problem can be formulated as a Markov Decision Process (MDP), defined by  $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$ , where  $\mathcal{S}$  and  $\mathcal{A}$  are the state space and action space, respectively.  $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$  is the transition probability distribution, and  $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$  is the reward function, which is discounted by  $\gamma \in [0, 1)$

**Algorithm 1** Iterative code generation

```

Input: message, trialmax
Output: code
1: trial ← 0
2: code ← Coder(message)
3: error ← run(code)
4: while error is not None do
5:   if trial > trialmax then
6:     return "Failed to generate code within trialmax"
7:   end if
8:   guidance ← Debugger(code, error)
9:   code ← Coder(guidance)
10:  error ← run(code)
11:  if error is None then
12:    error ← Evaluator(code)
13:  end if
14:  trial ← trial + 1
15: end while
16: return code

```

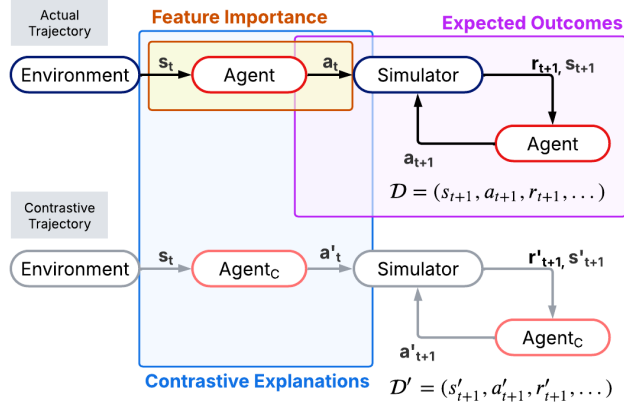


Figure 2: Explanatory scope of different XRL query types.

in temporal dimension. Within MDP, the goal of the RL agent is to learn a policy that maximizes the expected sum of discounted rewards.

In this work, we assume a policy network (actor) has already been trained to return action value from state variables, which is typical in policy-based and actor-critic methods. For demonstration purposes, we adopt the Soft Actor Critic (SAC) algorithm (Haarnoja et al., 2018) as our illustrative example. While we adopt SAC in our experiments, the proposed approach is readily generalizable to other policy-based or actor-critic algorithms.

3.5 XRL TOOLS

The central question in XRL is often framed as “Why did the agent execute this action?”. Since RL involves sequential decision-making with its goal of maximizing cumulative rewards, it offers multiple perspectives for interpreting agent behavior (Milani et al., 2024). As illustrated in Figure 2, we categorize the interpretation of RL agents into three perspectives: *Feature Importance* (FI), *Expected Outcomes* (EO), and *Contrastive Explanations* (CE). We limit our scope to local explanations and leave *policy-level* explanations for future work, to maintain the focus on specific state-action pairs.

To address each of these XRL tasks, we organize a set of XRL tools designed to interpret the local behavior of RL agents. These tools are automatically selected and called by LLM agents with contextually relevant arguments, enabling a more user-centric explanation.

270 **1) Feature Importance (FI)** approaches aim to identify which aspects of the current state most  
 271 influence the agent’s specific action. In our framework, we adopt DeepSHAP (Deep SHapley Addi-  
 272 tive exPlanations) (Lundberg & Lee, 2017), a post-hoc explainer tailored for deep neural networks,  
 273 to estimate the contribution of each state variable to the agent’s action value. DeepSHAP can effi-  
 274 ciently calculate feature attributions for the entire neural network by leveraging the chain rule, where  
 275 detailed calculation procedure is elaborated in the Supplementary materials.

276  
 277 **2) Expected Outcome (EO)** approaches aim to explain an agent’s behavior by analyzing antici-  
 278 pated future trajectories or rewards as a result of executing a particular action. EO explanation is  
 279 especially valuable in domains where rewards are delayed, or instant inverse response could happen.

280 In this work, we decompose the agent’s Q-value, the expected discounted sum of future rewards.  
 281 The Q-value can be decomposed in two complementary ways: (i) *component-wise*, which splits the  
 282 total reward into meaningful physical objectives, and (ii) *temporally*, which breaks Q-value across  
 283 time to identify when specific rewards are expected.

284 For component-wise decomposition, we follow the **code generation** scheme outlined in Algorithm  
 285 1. Given the Python code of the reward function used in RL training, the coder agent is instructed to  
 286 first identify the semantic components of the reward and rewrite the function to return the decom-  
 287 posed sub-rewards for each component.

288 For the temporal dimension, we adopt a strategy inspired by van der Waa et al. (2018), which used  
 289 a simulation model to generate the most probable future trajectory conditioned on the current state  
 290 and action. Although this approach assumes access to a simulation model, which may not always  
 291 be available in real-world applications, we adopt this approach since it can offer post-hoc future  
 292 trajectories without retraining the RL agent unlike other approaches (Ruggeri et al., 2025; Towers  
 293 et al., 2024). A resulting decomposition of Q-values is presented in Equation 1, where  $k$  denotes  
 294 component index,  $K$  represents total number of components, and  $\gamma$  is a discount factor.

$$295$$

$$296$$

$$297 Q^\pi(s, a) = \mathbb{E} \left[ \sum_{k=1}^K \sum_{t=0}^{\infty} \gamma^t r_{t,k} | s_0 = s, a_0 = a, \pi \right] \quad (1)$$

$$298$$

$$299$$

300 **3) Contrastive Explanations (CE)** explanations aim to answer contrastive questions such as  
 301 “What if?” or “Why not?”, highlighting why the agent selected the current action over plausible  
 302 alternatives. The key limitation of the previous XRL studies on contrastive explanation is that they  
 303 cannot capture high-level demands beyond simple actions, which restrict their applicability (Gajcin  
 304 & Dusparic, 2024). To enhance the flexibility of contrastive reasoning in RL practices, we introduce  
 305 three novel types of contrastive explanations.

306 **Action-based Contrasts (CE-A)** approach is conceptually similar to those used in previous XRL  
 307 studies, where the alternative scenario poses contrastive actions for a certain timestep (e.g., “Why  
 308 don’t we take action  $b$ , instead of action  $a$  at time  $t$ ?”). In our approach, we extend this concept by  
 309 allowing alternative actions to be kept over a specified time interval, providing richer insights into  
 310 how sustained action changes influence the system’s behavior.

311 **Behavior-based Contrasts (CE-B)** explanations construct scenarios from qualitative descriptions  
 312 about agent behavior. This type of contrastive explanation is particularly useful in control domains,  
 313 where alternative scenarios are often expressed using terms like *aggressive* or *conservative* without  
 314 specifying exact numeric details. While this resembles the tuning process of a PID (Proportional-  
 315 Integral-Derivative) controller, there are no explicit parameters corresponding to such behaviors for  
 316 RL agents. Therefore, it is necessary to interpret the nuance of the user’s query and translate it into  
 317 appropriate numerical adjustments.

318 To achieve this, we introduce *smoothing factor*  $\alpha$  that adjusts the overall behavior of the action  
 319 trajectory. For terms like *aggressive* or *conservative* trajectory, the idea of Polyak averaging (Polyak  
 320 & Juditsky, 1992) is employed. The action values of the adjusted trajectory ( $a_t^{adj}$ ) can be derived  
 321 by combining the values of the original trajectory ( $a_t^{orig}$ ), as shown in Equation 2. It modulates the  
 322 sensitivity of a trajectory to temporal changes by adjusting the value of  $\alpha$ . For example, *aggressive*  
 323 trajectories, which respond to temporal changes abruptly, are assigned  $\alpha > 1$ , while *conservative*  
 trajectories, where the action values evolve gradually, are set to  $0 < \alpha < 1$ . Semantically similar

terms (e.g., *calm*, *bold*, or *steady*) are also given appropriate  $\alpha$  values, owing to LLM’s robustness in linguistic nuances. For *opposite* behaviors, Equation 3 is used instead, as values may diverge if we directly use  $\alpha < 0$  in Equation 2.

$$a_t^{adj} = a_{t-1}^{adj} + \alpha(a_t^{orig} - a_{t-1}^{adj}) \tag{2}$$

$$a_t^{adj} = a_0^{orig} + \alpha(a_t^{orig} - a_0^{orig}) \tag{3}$$

**Policy-based Contrasts (CE-P)** addresses a broader question of how a fundamentally different control strategy would affect future trajectories, rather than just a localized action deviation. For example, a user might want to explore whether a simple on-off controller—one that operates between only two discrete values conditioned on state variables— could serve as an alternative to a RL policy, and compare their outcomes.

To support such queries, we employ the **code generation** scheme in Algorithm 1 to generate a new executable policy code that faithfully reflects the user’s intent. The generated policy is then executed in forward simulations to generate the rollout data.

In this work, we utilize generated contrastive scenarios to get insights from comparing the future state variables and rewards. Notably, those contrastive scenarios can also be used in conjunction with EO approaches, by comparing the expected decomposed rewards of contrastive trajectories with those of the original policy.

## 4 EXPERIMENTAL RESULTS

### 4.1 QUADRUPLE TANK SYSTEM

We evaluated our approach on the quadruple tank system, a well-known benchmark for RL-based control (Johanson, 2002). As illustrated in Figure 3, the system consists of four interconnected tanks, where water flows from two upper tanks to two lower tanks via gravity. The primary objective is to maintain specific water levels in the lower tanks ( $h_1, h_2$ ) by tracking desired setpoints—predefined target values specified by the operators. The pumped water is distributed to different tanks, introducing strong coupling between state variables within the system. This coupling induces potential competition between multiple reward components, making the system a suitable benchmark for evaluating XRL approaches.

The action variables are the voltages applied to two pumps ( $v_1, v_2$ ), which directly influence water flow. The state variables encompass the water levels in all four tanks, as well as the errors between the lower tank levels and their respective setpoints. The reward function is defined as the negative sum of the squared errors between the current  $h_1, h_2$  values and their setpoints, where a minor penalty component is included to discourage abrupt changes in the action variables. Although our approach is demonstrated in the context of process control, it can be readily generalized to other domains involving sequential decision making as long as a policy network of the agent is made available.

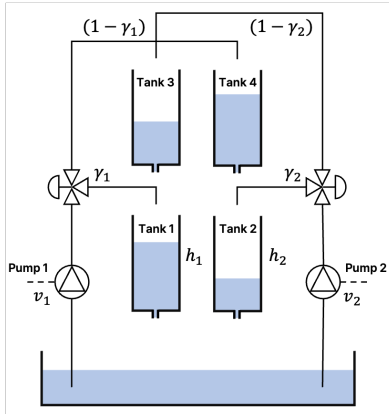


Figure 3: Illustration of quadruple tank system benchmark

### 4.2 SETTINGS

For training the RL models, we employed Stable-Baselines3 (Raffin et al., 2021), while the quadruple tank benchmark was simulated using the PC-gym package (Bloor et al., 2024). During training, the setpoints of the control objectives were periodically shifted to ensure that the learned policy could effectively track different values of setpoints. After training, we obtained a SAC agent capa-

ble of accurately regulating the setpoints. Details about environmental parameters and the training process of the SAC agent are reviewed in the supplementary material.

In this study, our experiments were designed to address the following three key research questions:

**RQ1:** Can *TalkToAgent* accurately interpret user queries and call relevant XRL functions with correct arguments?

**RQ2:** Can *TalkToAgent* reliably generate contrastive scenarios that reflect user’s intention?

**RQ3:** Can *TalkToAgent* correctly interpret the XRL results with domain-specific implications?

### 4.3 TASK CLASSIFICATION ACCURACY

To address RQ1, we first evaluated the coordinator’s ability in classifying user queries into the appropriate XRL function types. We generated a total of 90 example queries, comprising 20 queries for FI, EO CE-A, CE-B each, and 10 queries for CE-P type. We compared task classification accuracy of four different GPT variants (GPT-4.1, GPT-4o, GPT-4.1-mini, GPT-4.1-nano) under two prompt settings: *zero-shot*, where only function descriptions are given, and *few-shot*, where a few labeled examples are appended. The experiment was repeated 10 times independently for consistency.

Table 2: Task classification accuracy (%) across different GPT models in zero-shot and few-shot settings.

Setting	few-shot	zero-shot
gpt-4.1	<b>97.5 ± 0.9</b>	96.6 ± 0.8
gpt-4o	96.6 ± 0.3	95.7 ± 0.3
gpt-4.1-mini	93.3 ± 0.0	91.8 ± 0.9
gpt-4.1-nano	88.7 ± 1.1	85.0 ± 1.0

The results in Table 2 suggest that GPT-4.1 with few-shot prompting achieved the highest accuracy in task classification. Other models other than GPT-4.1-nano also effectively mapped user queries to the appropriate tasks, with accuracies higher than 90%. Furthermore, providing only function descriptions proved sufficient for correctly identifying the query type, although with a slight reduction in performance. Notably, most inaccuracies are mainly observed in CE-B queries, such as “keep the same action”, which could also have been handled by CE-A or CE-P solutions.

Moreover, the GPT-4.1 model achieved 100% accuracy in extracting the correct function arguments, including mapping qualitative behavior terms to proper numerical  $\alpha$  values in CE-B queries. The results ensure reliable selection of appropriate XRL tools in response to natural language queries.

### 4.4 CONTRASTIVE POLICY GENERATION ACCURACY

To address RQ2, we evaluated the fidelity of contrastive trajectory generations. In contrast to CE-A and CE-B queries where the intended trajectory is directly generated by the predefined functions, CE-P queries involve further code generation from LLM model, making it susceptible to generating hallucinated policies. This requires further validation to guarantee the fidelity of the resulting trajectories.

We tested 10 different CE-P queries by prompting *TalkToAgent* to generate corresponding policies. The maximum number of attempts ( $trial_{max}$  in Algorithm 1) was set to 10, and any inability to produce a valid policy within this limit was counted as a failure. To ensure consistency, the experiment was repeated across 7 independent iterations. Two GPT models (GPT-4.1, GPT-4o) were tested, and an ablation study with the debugger agent was further studied. For each setting, we measured both the total number of attempts required and failure counts.

The ablation study in Figure 4a indicates that the use of debugger significantly reduce both the average number of attempts and total failures. To further understand how errors evolve during iterative refinement, we tracked all errors that the agent encountered during contrastive policy generation. These errors were then classified by their type and content, and the results are illustrated as transition matrices in Figure 4b and c, which summarize the types of error transitions observed across all iterations of the *coder-evaluator-debugger* loop. The details of error types and their classification are in the supplementary material. We labeled an attempt as *success* if the agent generated contrastive policy code without any errors or hallucinations, and *failure* if it failed to generate code after 10

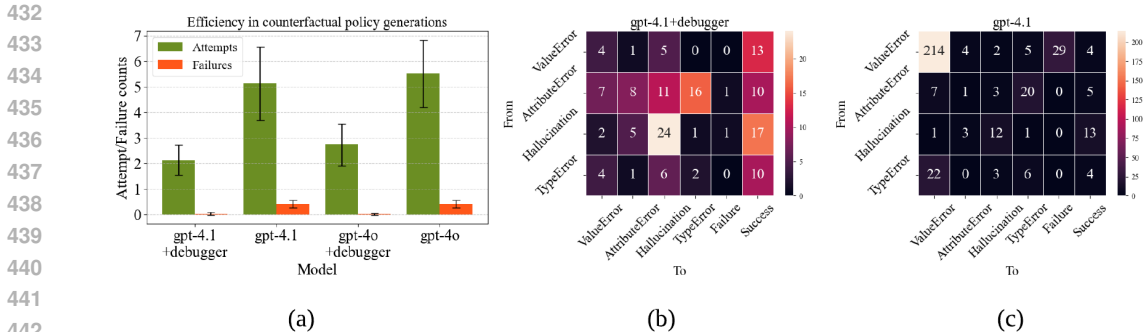


Figure 4: (a) Average total iterations and failure counts for each LLM model. (b,c) Error transition matrices showing how errors evolve over iterative trials for the GPT-4.1 model—(b) with the debugger, and (c) without.

trials. We counted all transitions between error types for 10 queries over all 7 iterations. The results show that without the debugger, the system often became stuck on recurring errors, especially on ValueError. In contrast, the debugger guided the agent toward a successful policy despite occasional hallucinations, which were also eventually resolved. This demonstrates the debugger’s effectiveness in providing meaningful guidance that helps the code converge to a valid solution.

#### 4.5 QUALITATIVE EVALUATION OF EXPLANATIONS

To address RQ3, we qualitatively evaluated the multimodal explanations generated for FI, EO, and CE queries. For each query, both XRL plots and corresponding text explanations were generated. Full explanations for all queries discussed in this paper are provided in the supplementary material.

First, for FI queries, the explanation revealed which state variables most strongly influenced each action variable. For example, pump  $v_1$  was primarily driven by the error term of  $h_2$ , while pump  $v_2$  was most influenced by the error term of  $h_1$ . For EO queries, the explanation highlighted how the agent prioritized rewards over time. We can notice that the agent initially focused on minimizing the  $h_2$ -tracking reward and later reduced both tracking rewards simultaneously. Lastly, for CE-B queries, an *opposite* control behavior to mitigate the inverse response of  $h_1$  is compared with the original trajectory. The resulting state and reward trajectories revealed that while the inverse response in  $h_1$  was alleviated instantly, it amplified the inverse response in  $h_2$ , ultimately degrading the overall control performance. Importantly, this phenomenon was well explained in relation to the coupling between control variables.

Overall, these results demonstrate that the system was able to generate accurate and intuitive explanations for diverse natural language queries by combining XRL visualizations with domain-aware contexts.

### 5 CONCLUSION

In this work, we presented *TalkToAgent*, a multi-agent LLM framework that bridges the gap between complex RL policies and domain experts with natural language interactions. Through experiments on the quadruple-tank system, we demonstrated its ability to accurately map user queries to suitable XRL tools, generate contrastive scenarios with reduced failure, and provide concise but accurate multimodal explanations.

A promising future direction is to extend our contrastive explanations to a counterfactual approach. While our current approach requires users to explicitly specify the contrastive actions, behaviors, or policies, a counterfactual approach could infer relevant modifications to actions to achieve a user’s desired outcome. This would answer high-level questions such as “What would happen if actions are adjusted to concentrate solely on maximizing a single reward component?” without specifying low-level actions. Furthermore, extending our approach to value-based networks and conducting case studies to various domains would be a promising future work.

## REFERENCES

- 486  
487  
488 Yotam Amitai, Yael Septon, and Ofra Amir. Explaining reinforcement learning agents through  
489 counterfactual action outcomes. In *Proceedings of the AAAI Conference on Artificial Intelligence*,  
490 volume 38, pp. 10003–10011, 2024.
- 491 Shahin Atakishiyev, Mohammad Salameh, Hengshuai Yao, and Randy Goebel. Explainable artificial  
492 intelligence for autonomous driving: A comprehensive overview and field guide for future  
493 research directions. *IEEE Access*, 2024.
- 494 Akanksha Atrey, Kaleigh Clary, and David Jensen. Exploratory not explanatory: Counterfactual  
495 analysis of saliency maps for deep reinforcement learning. *arXiv preprint arXiv:1912.05743*,  
496 2019.
- 497 Maximilian Bloor, José Torracca, Ilya Orson Sandoval, Akhil Ahmed, Martha White, Mehmet Mercangöz,  
498 Calvin Tsay, Ehecatl Antonio Del Rio Chanona, and Max Mowbray. Pc-gym: Benchmark  
499 environments for process control problems. *arXiv preprint arXiv:2410.22093*, 2024.
- 500 Jasmina Gajcin and Ivana Dusparic. Redefining counterfactual explanations for reinforcement learning:  
501 Overview, challenges and opportunities. *ACM Computing Surveys*, 56(9):1–33, 2024.
- 502  
503 Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy  
504 maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. Pmlr, 2018.
- 505  
506 Bradley Hayes and Julie A Shah. Improving robot controller transparency through autonomous  
507 policy explanation. In *Proceedings of the 2017 ACM/IEEE international conference on human-robot interaction*, pp. 303–312, 2017.
- 508  
509 Sandy H Huang, Kush Bhatia, Pieter Abbeel, and Anca D Dragan. Establishing appropriate trust  
510 via critical states. In *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 3929–3936. IEEE, 2018.
- 511  
512 Karl Henrik Johansson. The quadruple-tank process: A multivariable laboratory process with an  
513 adjustable zero. *IEEE Transactions on control systems technology*, 8(3):456–465, 2002.
- 514  
515 Zhengxian Lin, Kim-Ho Lam, and Alan Fern. Contrastive explanations for reinforcement learning  
516 via embedded self predictions. *arXiv preprint arXiv:2010.05180*, 2020.
- 517  
518 Guiliang Liu, Oliver Schulte, Wang Zhu, and Qingcan Li. Toward interpretable deep reinforcement  
519 learning with linear model u-trees. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 414–429. Springer, 2018.
- 520  
521 Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.
- 522  
523 Prashan Madumal, Tim Miller, Liz Sonenberg, and Frank Vetere. Explainable reinforcement learning  
524 through a causal lens. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 2493–2500, 2020.
- 525  
526 Stephanie Milani, Nicholay Topin, Manuela Veloso, and Fei Fang. Explainable reinforcement learning:  
527 A survey and comparative review. *ACM Computing Surveys*, 56(7):1–36, 2024.
- 528  
529 Tim Miller. Explanation in artificial intelligence: Insights from the social sciences. *Artificial intelligence*, 267:1–38, 2019.
- 530  
531 Joonsoo Park, Hyein Jung, Jong Woo Kim, and Jong Min Lee. Reinforcement learning for process  
532 control: Review and benchmark problems. *International Journal of Control, Automation and Systems*, 23(1):1–40, 2025.
- 533  
534 ATD Perera and Parameswaran Kamalaruban. Applications of reinforcement learning in energy  
535 systems. *Renewable and Sustainable Energy Reviews*, 137:110618, 2021.
- 536  
537 Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM journal on control and optimization*, 30(4):838–855, 1992.
- 538  
539

- 540 Erika Puiutta and Eric MSP Veith. Explainable reinforcement learning: A survey. In *International*  
541 *cross-domain conference for machine learning and knowledge extraction*, pp. 77–95. Springer,  
542 2020.
- 543 Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dor-  
544 mann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine*  
545 *learning research*, 22(268):1–8, 2021.
- 547 Franco Ruggeri, Alessio Russo, Rafia Inam, and Karl Henrik Johansson. Explainable reinforcement  
548 learning via temporal policy decomposition. *arXiv preprint arXiv:2501.03902*, 2025.
- 549 Pedro Sequeira and Melinda Gervasio. Interestingness elements for explainable reinforcement learn-  
550 ing: Understanding agents’ capabilities and limitations. *Artificial Intelligence*, 288:103367, 2020.
- 552 Dylan Slack, Satyapriya Krishna, Himabindu Lakkaraju, and Sameer Singh. Explaining machine  
553 learning models with interactive natural language conversations using talktomodel. *Nature Ma-*  
554 *chine Intelligence*, 5(8):873–883, 2023.
- 555 Kinga Szatmári, Gergely Horváth, Sándor Németh, Wenshuai Bai, and Alex Kummer. Resilience-  
556 based explainable reinforcement learning in chemical process safety. *Computers & Chemical*  
557 *Engineering*, 191:108849, 2024.
- 559 Mark Towers, Yali Du, Christopher Freeman, and Timothy J Norman. Explaining an agent’s future  
560 beliefs through temporally decomposing future reward estimators. In *ECAI 2024*, pp. 2790–2797.  
561 IOS Press, 2024.
- 562 Jasper van der Waa, Jurriaan van Diggelen, Karel van den Bosch, and Mark Neerincx. Con-  
563 trastive explanations for reinforcement learning in terms of expected consequences. *arXiv preprint*  
564 *arXiv:1807.08706*, 2018.
- 565 Xinzhi Wang, Shengcheng Yuan, Hui Zhang, Michael Lewis, and Katia Sycara. Verbal explanations  
566 for deep reinforcement learning neural networks with attention on extracted features. In *2019 28th*  
567 *IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pp.  
568 1–7. IEEE, 2019.
- 570 Zhang Xi-Jia, Yue Guo, Shufei Chen, Simon Stepputtis, Matthew Gombolay, Katia Sycara, and  
571 Joseph Campbell. Model-agnostic policy explanations with large language models. *arXiv preprint*  
572 *arXiv:2504.05625*, 2025.
- 573 Herman Yau, Chris Russell, and Simon Hadfield. What did you think would happen? explaining  
574 agent behaviour through intended outcomes. *Advances in Neural Information Processing Systems*,  
575 33:18375–18386, 2020.
- 577 Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding dqns. In  
578 *International conference on machine learning*, pp. 1899–1908. PMLR, 2016.
- 579 Ke Zhang, Jun Zhang, Pei-Dong Xu, Tianlu Gao, and David Wenzhong Gao. Explainable ai in  
580 deep reinforcement learning models for power system emergency control. *IEEE Transactions on*  
581 *Computational Social Systems*, 9(2):419–427, 2021.
- 582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593

# SUPPLEMENTARY MATERIALS

## S1. DEEPSHAP CALCULATION DETAILS FOR FEATURE IMPORTANCE QUERIES

In our work, DeepSHAP is employed to address feature importance (FI) queries. The attribution of feature  $y_k$  for function  $h$  is calculated as follows:

$$m_{xf}^j = \frac{\phi_j(f, x)}{x_j - \mathbb{E}(x_j)} \quad (4)$$

$$m_{yh}^k = \sum_{j=1}^n m_{yg_j}^k m_{xf}^j \quad (5)$$

$$\phi_j(h, y) = m_{yh}^k (y_j - \mathbb{E}(y_j)) \quad (6)$$

where  $m_{xf}^j$  denotes a multiplier for feature  $x_j$  in function  $f$ , and  $\phi_j(f, x)$  represents the corresponding Shapley value. Since Shapley values can be easily computed for simple blocks such as linear layers and activation functions, DeepSHAP can efficiently calculate feature attributions for the whole network by leveraging the chain rule. It first derives multipliers for each building block using Eq. 4, then recursively applies Eq. 5 across all layers. The final Shapley value  $\phi_j(h, y)$  for each feature is obtained via Eq. 6.

## S2. EXPERIMENTAL SETTING DETAILS

### S2.1. PARAMETER FOR QUADRUPLE TANK SYSTEM

Table 3: Quadruple-tank environment parameters used for simulation

Parameter	Value / Description
Total simulation time ( $T$ )	8000 seconds
Number of steps ( $n_{steps}$ )	400
Time step ( $\Delta t$ )	20 seconds
Action variables	$v_1, v_2$
State variables	$h_1, h_2, h_3, h_4, error_{h1}, error_{h2}$
Action space (low)	[0.1, 0.1]
Action space (high)	[10, 10]
State space (low)	[0, 0, 0, 0, -0.6, -0.6]
State space (high)	[0.6, 0.6, 0.6, 0.6, 0.6, 0.6]
Setpoint range	Uniformly sampled in [0.1, 0.5] every 40 steps
Initial state	[0.141, 0.112, 0.072, 0.42, 0.0, 0.0]

Table 3 summarizes the simulation parameters used in the quadruple-tank control environment, including observation and action spaces, initial conditions, and setpoint generation scheme.

The reward at each timestep  $t$  is defined as the negative sum of squared normalized tracking errors for the two controlled tank levels ( $h_1, h_2$ ) and a control effort penalty term. Formally, the reward is given by:

$$r_t = - \left[ 100 (\hat{x}_{h_1,t} - \hat{s}_{h_1,t})^2 + 100 (\hat{x}_{h_2,t} - \hat{s}_{h_2,t})^2 + \|\hat{u}_t - \hat{u}_{t-1}\|^2 \right]$$

where  $\hat{x}_{h_k,t}, \hat{s}_{h_k,t}$  are the normalized heights and their setpoints of tank  $k \in \{1, 2\}$ , respectively, and  $\hat{u}_t$  and  $\hat{u}_{t-1}$  are the normalized control input vectors at time  $t$  and  $t-1$ , respectively.

### S2.2. RL MODEL TRAINING SETUP

Table 4 presents the hyperparameters used for training the reinforcement learning (RL) agent.

Table 4: RL training settings

Parameter	Value / Description
RL algorithm	SAC (Soft Actor-Critic)
Total training steps ( $n_{\text{train}}$ )	$10^6$
Learning rate	0.001
Discount factor ( $\gamma$ )	0.9

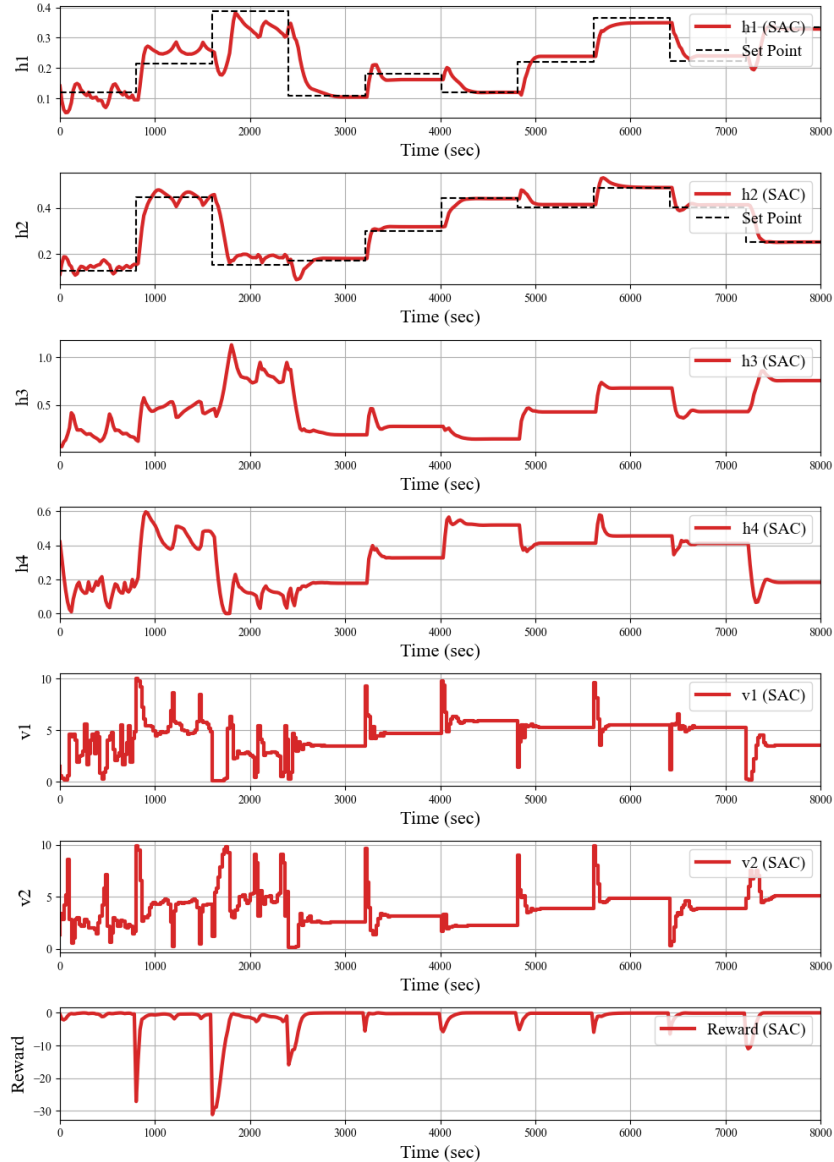


Figure 5: Rollout results of the trained SAC agent in the quadruple-tank system, showing tank water levels ( $h_1$ ,  $h_2$ ,  $h_3$ ,  $h_4$ ), pump voltages ( $v_1$ ,  $v_2$ ), and reward.

Figure 5 presents the rollout results obtained by the SAC agent trained with the above hyperparameters. The figure illustrates the water heights of the four tanks, the control inputs (pump voltages), and the corresponding reward values. As shown, the trained SAC agent exhibits slight oscillatory behavior during the initial phase, but after about 3,000 seconds, it successfully tracks the setpoints

for  $h_1$  and  $h_2$ . The reward remains close to zero for most of the rollout, except for instant drops when setpoints change abruptly.

Notably, inverse responses can be observed at certain time points, where the control action initially moves in the opposite direction of the setpoint change. For example, this behavior appears in  $h_1$  around 4,000 seconds and in  $h_2$  around 5,000 seconds. Such phenomena arise due to the coupling between action and state variables in the system dynamics. To highlight and analyze these behaviors, these instances are specifically selected for constructing example queries in this work.

### S3. PROMPTS USED FOR LLM AGENTS

To enable each of the five agents in the TalkToAgent framework to perform their designated roles effectively, we design specialized prompts tailored to the functionality and context of each agent. These prompts provide explicit instructions, structural constraints, and contextual descriptions, ensuring that each agent behaves consistently and accurately within the overall process control task. The following sections describe the detailed prompt formulations used for each agent.

#### S3.1. COORDINATOR

For the **Coordinator** agent, we incorporate the system description and environmental parameters to the prompt, to enable context-aware selection of appropriate XRL tools.

##### Coordinator prompt

Your task is to choose the next function to work on the problem based on the given function tools and user queries.

The brief explanation of control system is given below: `{system_description}`

Furthermore, the environment parameters are given below: `{env_params}`

Here are a few points that you have to consider while calling a function:

- When calling a function with `action` argument, make sure the action is within `env_params["actions"]`. Otherwise raise an error.
- When queried for a certain time interval, make sure to use the queried time itself when calling the function, without dividing by `delta_t` parameter.
- Also, don't scale neither state or action value, since it will be scaled at the subsequent functions.

#### S3.2. EXPLAINER

For the **Explainer** agent, we augment the prompt with both the system description and environmental parameters to ensure that XRL results can be interpreted within the correct system context. To enhance interpretability for end users, the prompt also explicitly includes descriptions of the specific XRL tool used (`{fn_description}`) and guidance on how to read the generated visualizations (`{figure_description}`). The contents of these placeholders vary depending on which XRL tool is selected. Additionally, we introduce the `max_token` parameter into the prompt, allowing users to flexibly adjust the level of detail by increasing or decreasing the token limit depending on their needs.

756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809

### Explainer prompt

You're an expert in both explainable reinforcement learning (XRL) and process control. Your role is to explain the user queries based on XRL results and related figures triggered by XRL functions.

**User query:** {user\_query}

Below are the name of the XRL function triggered and it's description:

- **Function name:** {fn\_name}
- **Function description:** {fn\_description}

- Also, for more clear explanation, the description of the system and its environment parameters are given as below:

- **System description:** {system\_description}
- **Environment parameters:** {env\_params}

- If XRL visualization are available, briefly explain how to interpret the all given visualization results.

- **Figure description:** {figure\_description}

#### Important Notes:

- If there are multiple agent actions to be explained, you will get sets of the plots. Make sure to interpret them individually.
- **IMPORTANT!** Make sure to relate the XRL results to input-output relationship within the system, based on the given system description.
- The explanation output must be concise and short enough (below {max\_tokens} tokens), because users may be distracted by too much information.
- Try to concentrate on providing only the explanation results, not on additional importance of the explanation.

### S3.3. CODER

The **Coder** agent is designed with separate prompts tailored to the purpose and context of each task. Specifically, two main tasks are supported: contrastive policy generation in response to policy-based contrast (CE-P) queries, and decomposed reward function generation for expected outcomes (EO) queries. To enable interactive debugging between the Coder and Debugger agents, the Coder also incorporates refining input messages that include debugging guidance passed from the Debugger agent via the `guidance` variable.

In addition, the generated code is instructed to include no additional comments or extraneous content. In order to address some occasional violations of this instruction we implement a separate `_sanitize` function to remove any non-code elements from the output prior to execution. This efficiently addresses issues such as the inclusion of markdown syntax (e.g., triple quotes) in decomposed reward functions, or inline comments in `CE_policy` classes.

#### S3.3.1. CONTRASTIVE POLICY GENERATION

For contrastive policy generation, the prompt explicitly provides a base structure for the `CE_policy` class, ensuring consistency with the overall format of the original RL policy. To further allow the Coder to match the expected output shape, an example output (`output_example`) from the original policy is also included in the prompt. The prompt also entails utility methods such as `self.env._scale_U(u)`, enabling the agent to use them when needed.

In our work, the generated policy is limited to simple, rule-based policies that can be encoded with basic logic. To enforce this constraint, the prompt instructs the agent to raise an error when queries request more advanced control methods such as MPC(Model Predictive Control) or PID(Proportional-Integral-Derivative) controllers. Extending the framework to support such methods is a promising direction for future work.

810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863

### Coder prompt – Contrastive policy generation

You are a coding expert that generates rule-based control logic, based on user queries. Your job is to write a code for the following Python class structure, named `CE_policy`:

```
=====
import numpy as np
np.random.seed(21)

class CE_policy():
def __init__(self, env, original_policy):
self.env = env
self.original_policy = original_policy

def predict(self, state, deterministic=True):
# INSERT YOUR RULE-BASED LOGIC HERE
return action
=====
```

Please consider the following points when writing the `predict` method:

- If the instruction requires you to modify the original policy, feel free to use the `self.original_policy.predict(state)` method.
- The output of the `predict` method (i.e., the action) should be within the range `[-1, 1]`, as it will be used by an external function that expects scaled values. You can scale the action values by using the method: `self.env._scale_U(u)`, if needed.
- The input `state` is also scaled. Ensure that your if-then logic works with scaled variables. To scale raw state values, you may use: `self.env._scale_X(x)`.
- The input for the `predict` method (`state`) is the same shape as the initial state `x0`.
- The output for the `predict` method (`action`) should match the shape of the output from the original policy. Example output: `{output_example}`
- If your code requires any additional Python modules, make sure to import them at the beginning of your code.
- Only return the code of the `CE_policy` class, **without any additional comments**.
- If the user requested controllers other than rule-based ones (e.g., MPC, PID), trigger the `raise_error` tool.

For accurate policy generation, here are some descriptions of the control system: `{system_description}`

Also, environment parameters used in process control: `{env_params}`

You will get a great reward if you correctly generate the contrastive policy function!

In the refinement stage, the input message incorporates both the error (or hallucination Exception raised by the Evaluator agent) encountered during policy execution and the corresponding debugging guidance generated by the Debugger agent. This information is provided to the Coder agent to support accurate correction of the previously generated policy. Importantly, the same constraints enforced during the initial generation—such as returning only the `CE_policy` class without any additional content—are emphasized in the refining input message to ensure consistency in output format.

### Coder input – Refine the contrastive policy with guidance

You previously generated the following code for a contrastive policy: `{self.prev_codes[-1]}`

However, the following error occurred during simulation: `{error_message}`

In order to debug this error, our Debugger agent suggested the following guidance: `{guidance}`

Please revise the code to fix the error. Only return the corrected `CE_policy` class. Also, you must still follow the original instructions from the initial prompt when modifying the code.

### 864 S3.3.2. DECOMPOSED REWARD GENERATION

865  
866 For decomposed reward generation, the prompt instructs the Coder agent to first interpret a given  
867 reward function and decompose the original scalar reward into multiple component-wise rewards.  
868 The agent is expected not only to generate the corresponding Python function that returns a tuple of  
869 individual rewards, but also to assign meaningful labels to each component based on their control  
870 objectives.

871 To facilitate semantic interpretation, the prompt explicitly requires the agent to output both the code  
872 and the list of concise component names, separated by a `\n---\n` delimiter. This formatting al-  
873 lows the two outputs to be programmatically parsed and used separately. Additionally, the system  
874 description is included in the prompt to help the agent generate physically and contextually appro-  
875 priate component names.

#### 876 Coder prompt – Decomposed reward generation

877  
878 Your job is to decompose reward function into multiple components. You will get a Python code of  
879 the reward function used to train the RL controller agent, and your job is to return its corresponding  
880 decomposed reward function.

881 Here are some requirements help you decompose the reward:

- 882 1. While the original reward function gives scalar reward, the decomposed reward should be in  
883 tuple format, which contains each component reward.
- 884 2. When returning answer, please only return the following two outputs:
  - 885 • The resulting Python function code. It would be better if necessary Python packages  
886 are imported. Remove unnecessary strings like `'''`, or `''' python`.
  - 887 • List of concise names of each control objective component. The format should be a  
888 Python list, so that we can directly translate it into a list object using the `ast` module.

889 These two outputs should be separated by the separating signal `\n---\n`.

- 890 3. You will also be given a brief description of the system. Please follow the description to  
891 appropriately decompose the reward.
- 892 4. The function's name should be in the form of  
893 `(original_function_name)_decomposed`.

894  
895 Here are the description of the current system and corresponding environment parameters that may  
896 help you decompose the reward function:

897 **System description:** `{system_description}`

898 **Environment parameters:** `{env_params}`

899 You will get a great reward if you correctly decompose the reward!

900  
901 The refinement process for decomposed reward functions closely mirrors that of policy generation.

#### 902 Coder input – Refine the decomposed reward with guidance

903  
904 You previously generated the following code for a decomposed reward function:  
905 `{self.prev_codes[-1]}`

906 However, the following error occurred during simulation: `{error_message}`

907 In order to debug this error, our Debugger agent suggested the following below: `{guidance}`

908 Please revise the code to fix the error. Only return the corrected `(original function`  
909 `name)_decomposed` function. Also, you still have to follow the instructions from the initial prompt  
910 when modifying the code.

### 913 S3.4. EVALUATOR

914  
915 The **Evaluator** agent is instructed to assess whether the generated contrastive policy faithfully re-  
916 flects the user's intention. If not, the agent is required to raise an error. As with other agents,  
917 the prompt includes the system description and environmental parameters to support context-aware  
evaluation.

918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971

**Evaluator prompt**

You are an evaluator that determines whether the Coder agent has correctly implemented the contrastive policy. If you believe the resulting trajectory faithfully follows the intention of the user message, you may confirm the policy. Otherwise, you must raise an error by calling the `raise_error` tool.

For accurate evaluation of the trajectory, here are some descriptions of the control system: `{system.description}`

Also, environment parameters used in process control: `{env.params}`

You will get a great reward if you correctly filter out any incorrectly generated contrastive policies!

### S3.5. DEBUGGER

The **Debugger** agent generates a guidance message for the Coder agent, suggesting how to fix the current code based on the encountered error or hallucination exception. This guidance is informed by the provided system description and environmental parameters.

**Debugger prompt**

You are a coding expert that provides proper guidance to the Coder agent to help refine the generated code and resolve the encountered error.

For accurate debugging, here are some descriptions of the control system: `{system.description}`

Also, environment parameters used in process control: `{env.params}`

You will get a great reward if you correctly guide the Coder agent to refine the policy accurately!

## S4. ERROR CATEGORIZATION IN CONTRASTIVE POLICY GENERATION

To classify the errors raised during contrastive policy generation, we first conducted clustering on collected error messages. Each error string was embedded using OpenAI’s `text-embedding-3-small` model to obtain vector representations. We then applied K-Means clustering to group the errors into 6 clusters. The result, visualized in Figure 6, was obtained by reducing the embedding dimension to two using Principal Component Analysis (PCA) method.

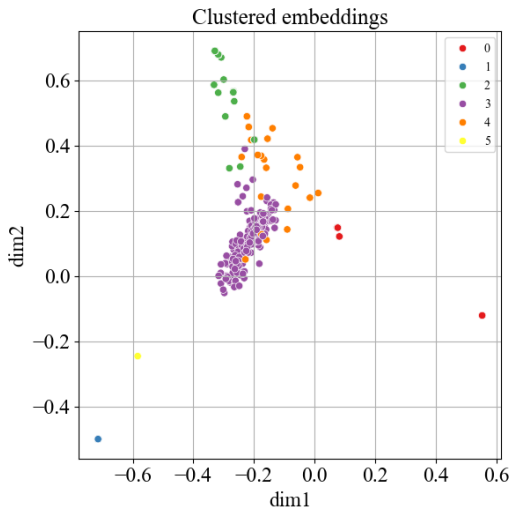


Figure 6: Clustering results of text embeddings of error messages collected over 7 iterations of contrastive policy generation experiments.

After clustering, we manually examined the error messages within each cluster to assign meaningful error categories, which are summarized in Table 5. We can observe that the error messages are generally clustered well according to their semantic type. Based on these categories, we analyze how errors evolved across iterations of the contrastive policy generation process and constructed a transition matrix as discussed in the main text.

Table 5: Representative error messages grouped by error category

Error Category	Representative Error Messages
ValueError	<ul style="list-style-type: none"> <li>setting an array element with a sequence. The requested array has an inhomogeneous shape after 1 dimensions.</li> <li>operands could not be broadcast together with shapes (4,) (6,)</li> <li>too many indices for array: array is 1-dimensional, but 2 were indexed</li> </ul>
AttributeError	<ul style="list-style-type: none"> <li>'make_env' object has no attribute 'o_space'</li> <li>'make_env' object has no attribute 'unscale_X'</li> <li>make_env.step() missing 1 required positional argument: 'action'</li> </ul>
Hallucination	<ul style="list-style-type: none"> <li>The trajectory does not faithfully follow the user's intention because the action v1 was not consistently set to the maximum value whenever h1 &lt; 0.2 during the specified timestep range (4000 to 4400).</li> <li>The trajectory does not implement the hybrid policy as specified. When h4 &gt; 0.7, v2 should be set to 1.5, but the provided control actions do not reflect this.</li> </ul>
TypeError	<ul style="list-style-type: none"> <li>can only concatenate tuple (not "int") to tuple</li> <li>unsupported operand type(s) for +: 'NoneType' and 'int'</li> <li>float() argument must be a string or a real number, not 'NoneType'</li> </ul>
Failure	<ul style="list-style-type: none"> <li>Failed after multiple attempts.</li> </ul>
Success	<ul style="list-style-type: none"> <li>Code successfully generated. Rollout complete.</li> </ul>

## S5. MULTIMODAL EXPLANATION RESULTS FOR QUADRUPLE TANK CONTROL SYSTEM

We provide representative user queries and corresponding explanations for FI, EO, CE-A, CE-B, and CE-P tasks. In this work, a `max_token` value of 200 was used in this work. Higher value of `max_token` would allow for more detailed responses. All examples were generated using OpenAI's GPT-4.1 model.

Note that the format of the responses may vary across runs, as no fixed output schema is enforced. Additionally, due to the inherent stochasticity of large language models, responses may not be fully deterministic even when using the same seed and temperature settings. Nevertheless, the XRL results themselves are fully reproducible, as they are generated by predefined functions.

### S5.1. FEATURE IMPORTANCE (FI)

**Query:** Which state variable makes great contribution to the agent's decisions at t=4020?

**Answer:** At t=4020, the state variable `Error_h2` (the error in Tank 2 level) makes the greatest contribution to the agent's decision for action `v1`, with a strong positive effect (+5.8). For action `v2`, `Error_h1` (the error in Tank 1 level) has the largest (negative) impact (-1.35). This means the agent is primarily adjusting v1 to correct Tank 2's deviation from setpoint, and v2 to correct Tank 1's deviation. The magnitude and direction of these contributions reflect the agent's focus on minimizing the most significant errors in the lower tanks, which are the main control targets in the four-tank system.

1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036

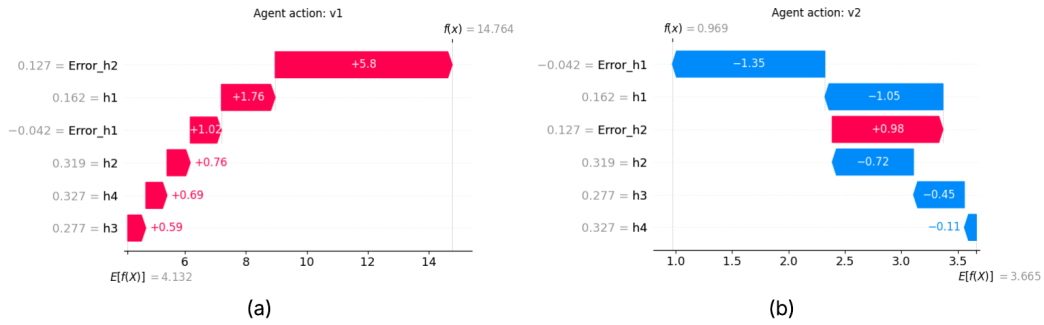


Figure 7: SHAP results for Feature Importance (FI) query

1037  
1038  
1039  
1040  
1041

## S5.2. EXPECTED OUTCOMES (EO)

1042

**Query:** What is the agent trying to achieve in the long run at  $t=4000$ ?

1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051

**Answer:** The agent’s action at timestep 4000 is primarily aimed at reducing the large tracking error for h2 (orange), as seen by the dominant negative reward for h2 tracking immediately after the action. Over the next steps, both h1 (green) and h2 tracking errors decrease, with h2 improving most rapidly. The agent sacrifices some control effort (blue) in the short term to quickly bring h2 closer to its setpoint, then gradually balances all objectives as both h1 and h2 tracking errors diminish and control effort is reduced. This reflects a prioritization of h2 tracking in the immediate future, then a return to balanced control.

1052  
1053

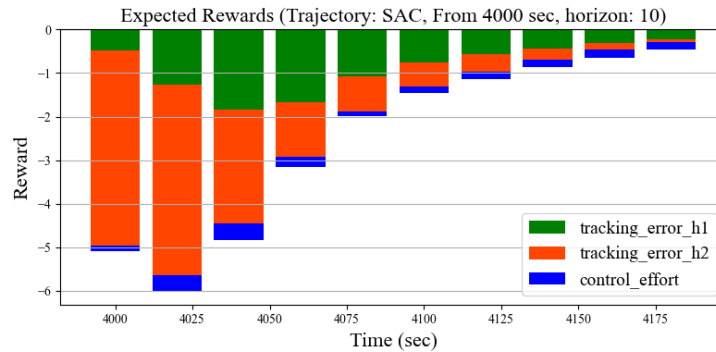


Figure 8: Decomposed reward results for Expected Outcome (EO) query

1064  
1065  
1066  
1067  
1068  
1069

## S5.3. ACTION-BASED CONTRAST (CE-A)

1070

**Query:** Why don’t we set the value of v1 action to 2.5 and v2 action to 7.5 from 4000 to 4200?

1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079

**Answer:** Setting  $v1=2.5$  and  $v2=7.5$  from 4000 to 4200 (purple, CF) caused h1 and h2 to deviate significantly from their setpoints, with h1 overshooting and h2 undershooting, compared to the actual controller (red) which kept both close to target. This is reflected in the much lower reward (more negative) for the counterfactual, indicating worse control performance. The fixed actions led to large, slow oscillations in tank levels (especially h3, h4), and a longer settling time. Thus, the original controller’s adaptive actions are superior for maintaining stability and setpoint tracking in this multivariable system.

1080  
 1081  
 1082  
 1083  
 1084  
 1085  
 1086  
 1087  
 1088  
 1089  
 1090  
 1091  
 1092  
 1093  
 1094  
 1095  
 1096  
 1097  
 1098  
 1099  
 1100  
 1101  
 1102  
 1103  
 1104  
 1105  
 1106  
 1107  
 1108  
 1109  
 1110  
 1111  
 1112  
 1113  
 1114  
 1115  
 1116  
 1117  
 1118  
 1119  
 1120  
 1121  
 1122  
 1123  
 1124  
 1125  
 1126  
 1127  
 1128  
 1129  
 1130  
 1131  
 1132  
 1133

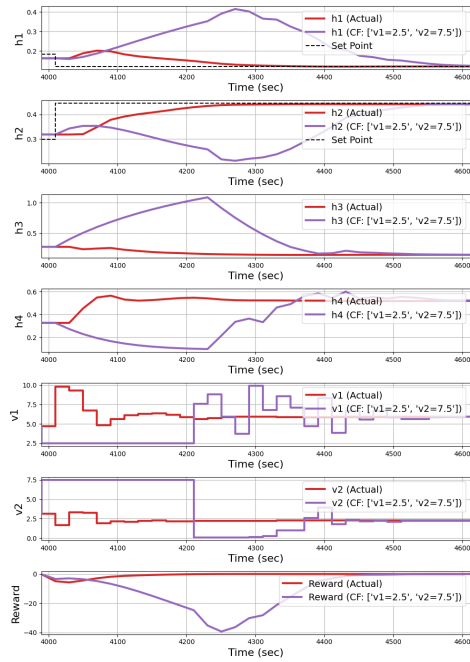


Figure 9: Forward simulation results for Action-based Contrast (CE-A) query

#### S5.4. BEHAVIOR-BASED CONTRAST (CE-B)

**Query:** Why don't we act opposite control from t=4000 to 4200, to constrain the instant inverse response in h1?

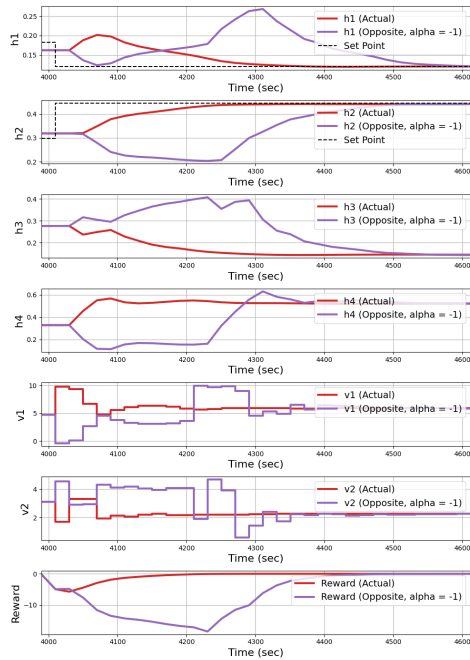


Figure 10: Forward simulation results for Behavior-based Contrast (CE-B) query

**Answer:** The contrastive (opposite) control from 4000–4200 (purple) causes  $h_1$  and  $h_2$  to deviate much further from their setpoints, with  $h_1$  showing a large overshoot and  $h_2$  a significant undershoot, compared to the actual control (red) which smoothly approaches the setpoints. This is due to the strong hydraulic coupling: applying the opposite action amplifies the inverse response, destabilizing both  $h_1$  and  $h_2$ . The reward plot confirms this, as the contrastive reward drops sharply (more negative), indicating worse performance and slower recovery. Thus, executing opposite control in this interval worsens both instant and long-term control, increasing settling time and error.

### S5.5. POLICY-BASED CONTRAST (CE-P)

**Query:** What would happen if we replaced the current RL policy with an on-off controller between 4000 and 4200 seconds, such that  $v_1 = 8.0$  whenever the error of  $h_1 < 0.0$ , and  $v_1 = 1.0$  otherwise; and similarly,  $v_2 = 8.0$  whenever the error of  $h_2 < 0.0$ , and  $v_2 = 1.0$  otherwise?"

**Answer:** From 4000 to 4200 sec, the on-off controller (purple) causes  $h_1$  to drop sharply below its setpoint, while  $h_2$  rises more quickly and overshoots its setpoint compared to the RL policy (red). This is due to the aggressive switching of  $v_1$  and  $v_2$  between 1.0 and 8.0, leading to oscillatory and less stable tank levels. The reward trajectory for the on-off controller is consistently lower (worse) than the RL policy, indicating higher error and poorer control performance. After 4200 sec, both policies converge, but the on-off controller results in more overshoot and slower settling. Overall, the RL policy provides smoother, more stable control with less overshoot and better reward.

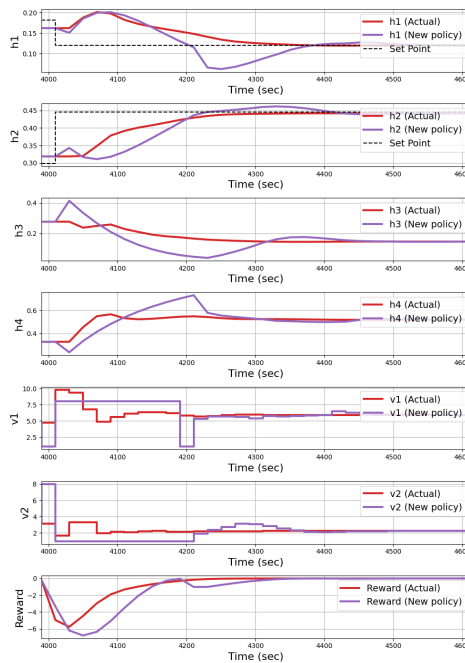


Figure 11: Forward simulation results for Policy-based Contrast (CE-P) query