Technical Report: Team SingaX for Embodied Agent Interface Challenge@NeurIPS 2025

Xinyuan Niu[1*] Zhiliang Chen[1] Vernon Yan Han Toh[2]

Yanchao Li[2] Zhengyuan Liu[3] Nancy F. Chen[3]

[1]National University of Singapore
[2]Nanyang Technological University
[3]Institute for Infocomm Research, A*STAR

Abstract

This work presents SingaX's approach to the Embodied Agent Interface Challenge, where we develop an LLM-driven pipeline for interpreting, decomposing, and executing natural language instructions in simulated household environments. Our methodology centers on leveraging large language models as semantic planners. A key innovation of our approach is a novel instruction induction framework that utilizes past error logging statements from development tasks to iteratively improve the LLM's ability to produce semantically consistent and logically correct actions. Our approach is training-free, cheap and efficient to run, and replaces manual effort required in crafting system prompts. In addition, we experimented with various other inference time verification and LLM aggregation approaches. In our report, *we also discussed and analyzed approaches that did not work well in the evaluation task*. Across the four challenge tasks—Goal Interpretation, Subgoal Decomposition, Action Sequencing, and Transition Modeling—we design task-specific prompt structures and cross-task validation routines that encourage coherent, executable outputs.

1 Introduction

Embodied agents operating in simulated or real-world household environments must reason over long-horizon natural language instructions, track intermediate states, and produce action sequences that are both semantically grounded and executable. Despite recent progress in large language models (LLMs), translating high-level objectives into precise low-level actions remains challenging due to ambiguities in instruction semantics, domain-specific constraints, and inconsistencies in multi-step reasoning [2]. In the **Embodied Agent Interface Challenge**, our work focuses on developing LLM-powered semantic planning pipelines that can propose natural language instructions in four structured challenges: Goal Interpretation, Subgoal Decomposition, Action Sequencing, and Transition Modeling.

Our work makes two novel contributions. The first stems from the key insight from our work is that is that errors in embodied task planning often stem not from a lack of knowledge, but from brittle prompt structures and insufficient exposure to system-level constraints during LLM inference. As a result, solutions from these LLMs often cannot be parsed effectively by existing verifiers, causing them to be marked as incorrect. To address this, we propose a novel instruction induction framework that incorporates historical error logs collected during development tasks. By feeding failure cases back into the prompting cycle as structured feedback, the LLM progressively learns to avoid prior logical inconsistencies and pre-emptively safeguards its responses in future unseen problems. This produces more reliable action plans without requiring model fine-tuning.

*Corresponding author: xinyuan@u.nus.edu

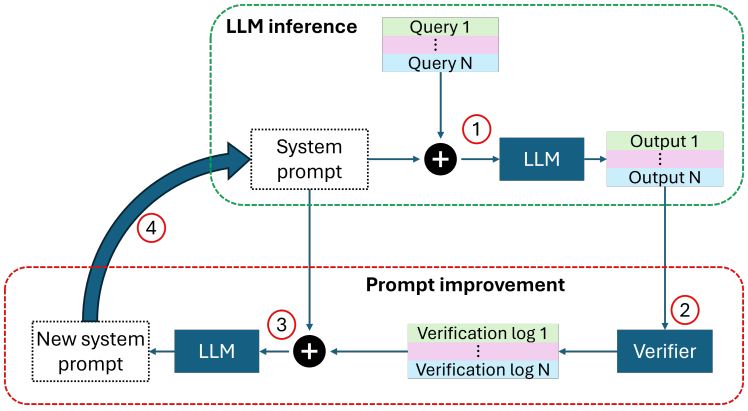

Figure 1: Overview of the iterative prompt induction framework. During development phase, output from the `Answering LLM` are evaluated with the provided verifiers to produce evaluation logs. These are fed into the `Safeguard Generation LLM` to produce a new set of system prompts. During the test phase, as the verifier is no longer available, the improved system prompt is directly used for generation without the prompt improvement block.

The second contribution is a **lightweight verification module** that validates the consistency of predicted subgoals and actions against environment constraints, filtering or revising plans before execution. Together, these contributions demonstrate a scalable and modular approach to embodied task planning, offering insights for integrating LLMs into interactive agent pipelines.

2 Related Work

Prompt induction enables LLMs to infer natural language instructions from few-shot examples, improving task generalization without fine-tuning [4, 3, 1]. Beyond generalization, prompt induction reveals the inherent capabilities of base models on specialized tasks by learning from failure cases: analyzing errors produced by vanilla prompts allows the model to synthesize refined instructions that better elicit its task-specific knowledge.

3 Method

From our preliminary experiments, we found that the LLM often give solutions that yield poor results. There are a few reasons why this occurs. First, the LLM solution often does not abide to the JSON format required by the submission verifier. As a result, solutions that are almost correct are often marked as incorrect due to parsing errors. Second, in many tasks, the LLM produces correct action sequences at first but then produces actions which are invalid in the given environment. For example, the LLM frequently tries to retrieve items inside closed containers, or requires the character move when the character is sitting or lying down.

3.1 Iterative Prompt Induction Framework

Our approach operates by generating error logs associated with the LLM solution generated on the training dataset by the initial provided system prompts (1 and 2 in fig. 1). We initially considered manually looking through the logs to identify the key mistakes made by the LLM (by eye and by code). However, this proved to be too time consuming, due to initial provided prompts performing quite poorly resulting in a large number of mistakes to sieve through. This gave us the idea of simply making use of an LLM to perform this task, as LLMs are known to be able to extract information and perform summarization well from long contexts. We integrate and refine these error logs into our LLM prompts (3 and 4 in fig. 1), generating guardrail statements and "emphasis rules" for the LLM

across different environments. Briefly,

1. During development phase, we use the standard prompts (given by the challenge organizers) with an `Answering LLM` to generate default solutions for the development scenarios in the challenge (1 in fig. 1). These solutions are then evaluated using the local evaluation pipeline provided by the organizers (2 in fig. 1. The evaluation pipeline produces error logs, indicating why a solution is incorrect - for instance, if our solution contains the action of moving a cup from the top of a book to the table but the cup currently is not on a book, then the error log would point out this error.
2. We pass these error logs into an `Safeguard Generation LLM` (3 in fig. 1) to reason about the most common error types faced during development and generate a list of pre-emptive safeguards that we include in our prompts (4 in fig. 1). For instance, if the LLM frequently creates invalid solutions because of missing a closing bracket in its JSON outputs, then the external LLM will create a pre-emptive safeguard: "When structuring your solution into a JSON file, make sure the number of closing brackets matches the number of opening brackets". We also ask the LLM to generate a new prompt that incorporates these safeguards into the original prompts given by the challenge organizers.
3. Using the new prompt that contains these safeguards, we pass it into the `Answering LLM` to generate the solutions for different action tasks during the evaluation phase. Because safeguards are emphasized in the new prompts, the LLM is less likely to make the same mistakes during evaluation. What is noteworthy is that these safeguards are *generalizable* to tasks scenarios not seen in the development phase. Hence, the optimized prompts generated from the development phase can be deployed effectively during the evaluation phase.

3.2 Multi-Model Best-of-N at Test Time

Besides prompt-induction–based optimization, we also investigate a test-time scaling strategy that improves solution quality without modifying model parameters. The baseline we consider is a standard Best-of-N (BoN) scheme: for each instance, a single base LLM is queried N times with stochastic decoding, and a verifier then selects the best candidate among these samples. While this approach can reduce random errors, the candidates are still drawn from essentially the same proposal distribution and often exhibit highly correlated failure modes.

Our method generalizes this idea from "more samples from one model" to "diverse samples from multiple models". Concretely, for each task instance we query three heterogeneous LLM families—Qwen, Gemini, and GPT—using the same task-specific prompt template described in Section 3. This yields a small but diverse candidate set

$$\mathcal{C} = \{c_{\text{Qwen}}, c_{\text{Gemini}}, c_{\text{GPT}}\},$$

where each element represents a complete JSON-formatted solution proposed by one model. Compared to repeatedly sampling from a single model, this multi-model design enlarges the effective proposal space at roughly the same total compute budget.

To choose the final output, we employ a strong LLM verifier as a scoring module. Given all candidates in $\mathcal{C}$, the verifier is prompted to assess (i) syntactic validity with respect to the required JSON schema, (ii) consistency with environment and action constraints, and (iii) plausibility of achieving the given goal. The verifier produces a scalar score for each candidate, and we return the solution with the highest score as the final prediction:

$$c^* = \arg\max_{c \in \mathcal{C}} \text{VerifierScore}(c).$$

In practice, we observe that cross-model diversity makes the verifier substantially more effective than in the single-model BoN setting, since different models tend to make different types of mistakes. As a result, this multi-model BoN strategy serves as a lightweight yet effective form of test-time compute scaling that is fully compatible with the overall pipeline in Section 3.

3.3 Critic Best-of-N

In addition, we propose a *Critic Best-of-N* (Critic BoN) framework to enhance generation quality through iterative refinement. The workflow initiates with an initial query q (representing the original goal and constraints) fed into a Generator model, which produces a set of n candidate outputs, denoted as $\{o_1, o_2, \ldots, o_n\}$.

Subsequently, each candidate is evaluated by a Critic agent. This agent validates the generated JSON against the original constraints using a specific checklist of instructions. The Critic outputs a structured JSON object comprising a `critique_summary` and a list of specific, actionable `issues`. If the `issues` list for any candidate c_i is empty, indicating a flawless output, that candidate is immediately returned as the final result.

If errors persist across all candidates, the system applies a Best-of-N selection strategy to identify the optimal candidate o^* by minimizing the number of identified issues:

$$o^* = o_k \quad \text{where} \quad k = \underset{i}{\operatorname{argmin}}(|c_i^{\text{issues}}|) \tag{1}$$

To facilitate iterative refinement, the selected best candidate and its associated critique summary (o^*, c^*) are fed back into the Generator. The model then produces N new refined outputs conditioned on this pair, repeating the cycle until the validation criteria are met.

3.4 Output processing

To prevent parsing errors of the generated output, we performed rejection sampling by checking if the generated output conforms to a JSON format. This is done during inference time as the LLM outputs are returned from the API, and any outputs that fail the check are re-queried up to a maximum of 3 times.

4 Insights and Discussion

We further analyze the safeguards produced by the `Safeguard Generation LLM` and observe that many of the resulting guardrails correspond to generalizable action-pattern rules that prevent common classes of errors. Rather than overfitting to specific development scenarios, these safeguards capture high-level principles that ensure state validity, enforce correct action ordering, and maintain consistent output formatting. In table 1, we summarize the key categories of safeguards identified across tasks by GitHub Copilot. Note that these are actual guardrails generated by the `Safeguard Generation LLM`.

These safeguards collectively contribute to improved robustness by constraining the LLM to produce logically consistent, state-valid, and structurally correct action plans across both development and evaluation scenarios. From our experimental results (Table 4), these guardrails are shown to be effective and more importantly, generalizable on the evaluation task.

5 Experiments

We evaluated our method on the development scenarios provided by the organizers, before moving onto applying it on the test scenarios (whose performance eventually appears on the leader board).

5.1 Iterative Prompt Induction Framework

For our leader board submission, we used the `Qwen3-235B-Thinking` model as our `Answering LLM`. For our `Safeguard Generation LLM`, we surprisingly found `GPT-5` via GitHub Copilot's agent mode to be effective at scanning the error logs and producing safeguards for our optimized prompts. For the logs to be more informative, we made minor modifications to provided source code to clean up and format the logs. Due to the long context length of `GPT-5` (128,000 tokens), it is able to ingest the entire log JSON file generated by the verifiers in the development loop, extract

Category	Guardrail
Spatial grounding	Always WALK to the specific object you act on (not just the room).
State preconditions	• If CLOSED: OPEN. • If PLUGGED_OUT and needs power: PLUGIN. • If faucet needed: SWITCHON faucet.
Action ordering	• Device use: WALK → (PLUGIN?) → SWITCHON → USE. • Cleaning: WALK → WASH → RINSE. • Drinking: WALK → GRAB → (POUR) → DRINK.
Avoid redundancy	Avoid duplicates and irrelevant actions; stop once the goal is satisfied.
Output format	Output strictly in the specified JSON schema.
PDDL rules	• The `:effect` lists the changes which the action imposes on the current state. The `:precondition` consists of predicates and 6 possible logical operators: `or`, `and`, `not`, `exists`, `when`, `forall`. • Effects should generally be several effects connected by `and` operators. • For each effect, if it is a conditional effect, use `when` to check the conditions. Semantics: `(when [condition] [effect])` means if the condition is true before the action, the effect occurs afterwards. • If it is not a conditional effect, use predicates directly. • The `not` operator negates a predicate, meaning the condition will not hold after the action is executed. • The `forall` operator is followed by a variable and a body. Format: `forall (?x - type) (predicate1 ?x)`, meaning for all objects of that type, the predicate holds.

Table 1: Some of the guardrails generated by the `Safeguard Generation LLM` (GitHub Copilot) to reduce errors when generating solutions for development scenarios. These are integrated into by the `Safeguard Generation LLM` into the new system prompts.

and reason about the key mistakes made by the `Answering LLM` and make use of tool calls to update the prompt. fig. 2 shows a sample section of the prompt and response by the `Safeguard Generation LLM`. Our submission used a single prompt induction loop (system prompt was only updated once by the `Safeguard Generation LLM`).

As shown in table 1, we noticed that the `Safeguard Generation LLM` would frequently classify similar problems and provide guidelines or templates for the `Answering LLM` to follow. It also provided checklists for the `Answering LLM` to verify against before the final output is produced. This resulted in the greatest performance gains across all tasks. However, we occasionally still had to manually edit some of the prompts to correct minor mistakes made by the `Safeguard Generation LLM` and account for nuances in the problem tasks. For instance, the `Safeguard Generation LLM` would occasionally copy extraneous or miss out copying over certain examples from the logs. However, utilizing the `Safeguard Generation LLM` drastically reduced the number of logs we had to manually look through and the amount of effort needed in crafting the prompts.

5.1.1 Cost analysis of prompt induction optimization

To optimize our prompts, we need to perform one round of answer generation (before using the error logs to optimize the prompts). Because we mostly used OpenRouter (`https://openrouter.ai/`) to generate outputs from `Qwen3-235B-Thinking`, it is not free to use the `Answering LLM` to generate the initial responses. Before we began our experiments, we estimated the cost needed to perform a single round of answer generation. Although other models such as `GPT-5`

might be able to perform better, we chose `Qwen3-235B-Thinking` as the `Answering LLM` as it is Open-source, and has a very competitive pricing on OpenRouter of $0.11 and $0.60 per million input and output tokens respectively[1]. As shown in table 2, only approximately $21 is required for the entire experiment (over 8000 prompts across development and test phases for all 4 tasks over 2 environments). In comparison, `GPT-5` costs about $1.25 and $10.00 per million input and output tokens respectively[2], and would have costed more than 10 times as much for the entire experiment.

Table 2: Estimated cost of optimizing prompts using our method. This assumes all queries successful, actual cost will be slightly higher due to failed generations. Output token length includes reasoning tokens. Cost for task reports the combined cost for all the prompts in the task.

Environment	Task	Dev loop			Test loop		
		Mean token length		Cost for	Mean token length		Cost for
		Prompt	Output	task ($)	Prompt	Output	task ($)
behavior	goal_interpretation	1216	5717	0.36	2119	5229	0.35
	subgoal_decomposition	2657	12014	0.75	3683	8851	0.58
	action_sequencing	3415	8188	0.53	3701	7298	0.50
	transition_modeling	3414	10399	0.66	3660	8352	0.57
virtualhome	goal_interpretation	1874	7706	1.65	2994	5513	5.46
	subgoal_decomposition	3056	7225	1.58	3571	8386	8.14
	action_sequencing	2256	5264	1.16	3336	5777	5.75
	transition_modeling	3635	9902	1.88	5398	8681	8.70
Total cost				3.75			17.41

5.1.2 Performance gains from optimized prompts

	behavior		virtualhome	
	Baseline	Ours	Baseline	Ours
goal_interpretation (f1)	85.4	86.2	37.8	64.5
subgoal_decomposition (task sr)	59.0	79.0	71.3	79.3
action_sequencing (task sr)	72.0	85.0	67.6	92.0
transition_modeling (f1)	58.1	98.9	45.1	99.5
transition_modeling (sr)	96.0	99.0	91.0	99.9

Table 3: Comparison against baseline results by `Host_84085_Team` on the leaderboard.

We show in table 3 our improvements from the optimized prompts beyond the baseline as reported by `Host_84085_Team` on the leaderboard. At the time of writing, our approach ranked second on the leaderboard, despite not requiring any training and does not seem to be overfitted to the development set of the behavior set.

To verify the performance gains from the optimized prompts, we evaluated our method over different smaller LLMs to verify its effectiveness. Table 4 shows our method consistently improves the task performance of different LLMs over all development tasks significantly even on the smaller LLMs.

Since we used the same development scenarios to generate the initial error logs and optimized prompts, *it comes at no surprise* that our optimized prompts would work well over the same development scenarios. For instance, if the development scenarios consistently requires us to move a cup from the top of a book to the table, and our `Answering LLM` constantly thinks that the cup is not

[1] https://openrouter.ai/qwen/qwen3-235b-a22b-thinking-2507/providers
[2] https://openrouter.ai/openai/gpt-5-chat/providers

on the book (which is an error), then our optimized prompt would safeguard against such common errors.

Table 4: Overview of results (%) on the evaluation phase. V: VirtualHome, B: BEHAVIOR.

	Goal Interpretation		**Action Sequencing**				**Subgoal Decomposition**				**Transition Modeling**				**Average Perf.**		**Overall Perf.**
	F_1		$TaskSR$		$ExecSR$		$TaskSR$		$ExecSR$		F_1		$PlannerSR$		$ModuleSR$		
Model	*V*	*B*	*V*	*B*	*V*	*B*	*V*	*B*	*V*	*B*	*V*	*B*	*V*	*B*	*V*	*B*	
Qwen 3 4B	23.9	39.7	58.4	39.0	67.0	56.0	54.9	43.0	79.1	54.0	30.3	35.9	43.5	46.0	43.53	40.66	42.09
(+ Optimized Prompt)	38.6	30.6	63.7	41.0	74.6	51.0	55.7	55.0	79.8	70.0	68.8	52.0	47.0	70.0	53.98	46.90	50.44
	(↑ 14.7)	(↓ 9.1)	(↑ 5.3)	(↑ 2.0)	(↑ 7.6)	(↓ 5.0)	(↑ 0.8)	(↑ 12.0)	(↑ 0.7)	(↑ 16.0)	(↑ 38.5)	(↑ 16.1)	(↑ 3.5)	(↑ 24.0)	(↑ 10.4)	(↑ 6.2)	(↑ 8.4)
Qwen 3 8B	23.4	69.7	58.4	44.0	69.5	58.0	58.9	40.0	81.3	50.0	38.3	53.7	80.2	73.0	49.99	54.26	52.13
(+ Optimized Prompt)	39.5	73.3	67.9	56.0	80.7	65.0	61.1	56.0	82.7	70.0	81.0	63.7	92.5	91.0	63.81	65.66	64.74
	(↑ 16.1)	(↑ 3.6)	(↑ 9.5)	(↑ 12.0)	(↑ 11.2)	(↑ 7.0)	(↑ 2.2)	(↑ 16.0)	(↑ 1.4)	(↑ 20.0)	(↑ 42.7)	(↑ 10.0)	(↑ 12.3)	(↑ 18.0)	(↑ 13.8)	(↑ 11.4)	(↑ 12.6)
Qwen 3 14B	24.8	71.0	66.0	46.0	82.1	58.0	62.5	45.0	82.1	53.0	43.0	59.6	63.5	44.0	51.64	53.45	52.54
(+ Optimized Prompt)	41.5	73.0	65.5	57.0	80.9	65.0	66.3	66.0	86.6	78.0	79.9	64.6	82.4	83.0	63.61	67.45	65.53
	(↑ 16.7)	(↑ 2.0)	(↓ 0.5)	(↑ 11.0)	(↓ 1.2)	(↑ 7.0)	(↑ 3.8)	(↑ 21.0)	(↑ 4.5)	(↑ 25.0)	(↑ 36.9)	(↑ 5.0)	(↑ 18.9)	(↑ 39.0)	(↑ 12.0)	(↑ 14.0)	(↑ 13.0)
Qwen 3 32B	28.0	65.5	63.2	59.0	77.3	71.0	65.9	47.0	86.3	55.0	45.4	62.5	76.2	78.0	54.48	60.44	57.46
(+ Optimized Prompt)	39.4	68.5	67.4	63.0	81.1	72.0	66.3	66.0	85.7	79.0	78.2	68.3	84.4	89.0	63.60	69.04	66.32
	(↑ 11.4)	(↑ 3.0)	(↑ 4.2)	(↑ 4.0)	(↑ 3.8)	(↑ 1.0)	(↑ 0.4)	(↑ 19.0)	(↓ 0.6)	(↑ 24.0)	(↑ 32.8)	(↑ 5.8)	(↑ 8.2)	(↑ 11.0)	(↑ 9.1)	(↑ 8.6)	(↑ 8.9)
Qwen 3 30B A3B	26.80	79.10	69.30	53.00	81.50	68.00	61.10	56.00	83.90	66.00	36.70	49.70	82.10	69.00	54.15	61.86	58.01
(+ Optimized Prompt)	42.80	69.70	70.00	54.00	83.60	65.00	64.30	73.00	86.10	88.00	75.60	61.50	92.60	88.00	65.30	67.86	66.58
	(↑ 16.00)	(↓ 9.40)	(↑ 0.70)	(↑ 1.00)	(↑ 2.10)	(↓ 3.00)	(↑ 3.20)	(↑ 17.00)	(↑ 2.20)	(↑ 22.00)	(↑ 38.90)	(↑ 11.80)	(↑ 10.50)	(↑ 19.00)	(↑ 11.15)	(↑ 6.00)	(↑ 8.6)

Next, we applied our method to the evaluation scenarios for the leader board submission. It is important to note that because the evaluation scenarios for the lead board submission cannot be run locally, we cannot generate error logs for them and consequentially, cannot specifically optimize our prompts towards these evaluation scenarios. Instead, we use the optimized prompts from the development scenarios and use it on the evaluation scenarios.

5.2 Critic BoN Experiments

Table 5: Overview of results (%) on BEHAVIOR for Critic BoN.

		Goal Interpretation	**Action Sequencing**		**Subgoal Decomposition**		**Transition Modeling**		**Average Perf.**
Model	Refinement	F_1	$TaskSR$	$ExecSR$	$TaskSR$	$ExecSR$	F_1	$PlannerSR$	
Qwen3-8B	0	73.47	56.00	64.00	65.00	75.00	62.91	86.00	67.23
	1	74.93	51.00	61.00	48.00	57.00	62.67	82.00	61.57
	2	75.07	54.00	61.00	46.00	62.00	62.85	81.00	61.75
	3	74.98	50.00	59.00	45.00	58.00	62.82	86.00	61.10
	4	75.08	52.00	58.00	45.00	56.00	62.78	85.00	61.49
Qwen3-14B	0	76.67	60.00	75.00	64.00	75.00	65.50	86.00	69.10
	1	70.48	58.00	66.00	57.00	67.00	65.79	80.00	64.60
	2	72.72	61.00	71.00	55.00	64.00	66.03	80.00	65.43
	3	72.44	61.00	69.00	58.00	68.00	65.96	82.00	66.35
	4	72.61	62.00	72.00	55.00	64.00	65.91	81.00	65.77

Experimental Setup. We evaluated the Critic BoN framework on the BEHAVIOR benchmark using two base models: Qwen3-8B and Qwen3-14B. The experiments were conducted with a candidate pool size of $N = 4$ and a maximum limit of 4 refinement loops per query. We assessed performance across all four tasks in BEHAVIOR: Goal Interpretation (F_1), Action Sequencing, Subgoal Decomposition, and Transition Modeling, reporting both Task and Execution Success Rates (SR) where applicable.

Results. Table 5 summarizes the results. In this configuration, the iterative refinement process did not yield consistent improvements over the baseline. For both model sizes, the initial generation (Refinement 0) generally achieved the highest average performance, with Qwen3-8B peaking at 67.23% and Qwen3-14B at 69.10%.

Subsequent refinement iterations resulted in a regression in metrics, particularly within Subgoal Decomposition, where Qwen3-8B dropped from 65.00% (Refinement 0) to 45.00% (Refinement 4). While Goal Interpretation remained relatively stable for the 8B model, the 14B model exhibited a decline from 76.67% to 72.61%. These results suggest that while the Critic mechanism identifies issues, the iterative loop with $N = 4$ may introduce instability or over-correction in this specific domain, preventing the generator from surpassing its zero-shot baseline.

In our final submission, we provided the results of the iterative prompt induction framework. Although it produced promising results, the BoN approach proved too expensive and time-intensive to run with our limited budget. Despite this, we believe it can be further integrated to further improve the performance of the overall framework.

6 Discussion

The safeguards produced by the `Safeguard Generation LLM` tend to capture high-level reasoning patterns rather than scenario-specific heuristics. This contributes to their ability to generalize to unseen evaluation tasks, even when the underlying error types arise in new forms. In effect, our approach teaches the LLM how to avoid classes of mistakes, not just how to correct specific instances. This contrasts with conventional fine-tuning or reward-based methods, which often overfit to the development data or require costly repeated training cycles.

Another important observation is that integrating safeguards directly into prompts leverages the LLM's inherent strengths: rule-following, adherence to explicit constraints, and improved self-consistency when guidance is clearly specified. By elevating common pitfalls to "emphasis rules," the system reduces syntactic, semantic, and state-consistency errors without modifying model weights or requiring computationally intensive retraining.

Overall, our findings suggest that error-log–guided prompt optimization is a lightweight yet powerful strategy for enhancing reliability. It provides a scalable path for improving LLM performance in structured decision-making tasks where correctness hinges on avoiding subtle logical or environmental inconsistencies. As LLMs continue to be deployed in increasingly complex domains, methods that transform error signals into generalizable procedural safeguards may offer an efficient alternative to model-level optimization.

7 Conclusion

Our method demonstrates that structured error-driven prompt refinement can substantially improve the robustness of LLM-based planners in embodied or procedural reasoning tasks. A key insight is that error logs—often treated as terminal outputs of an evaluation pipeline—can instead be repurposed as a valuable source of supervision. By systematically aggregating these logs and distilling them into explicit safeguard rules, we create an iterative feedback loop that strengthens the LLM's ability to avoid common failure modes.

Reproducibility Statement

Our experiments made use of open and closed sourced LLMs (`Qwen3-235B-Thinking` and `GPT-5` respectively). As described in section B, we used greedy sampling with a temperature of 0.0 for more deterministic experiments. Code and prompts used in the experiments will be published on our GitHub page. Only the development set was used in the development loop of the experiment, along with information provided by the verifier logs extracted from the provided source code.

References

[1] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2025.

[2] Manling Li, Shiyu Zhao, Qineng Wang, Kangrui Wang, Yu Zhou, Sanjana Srivastava, Cem Gokmen, Tony Lee, Erran Li Li, Ruohan Zhang, et al. Embodied agent interface: Benchmarking llms for embodied decision making. *Advances in Neural Information Processing Systems*, 37:100428–100534, 2024.

[3] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[4] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*, 2022.

A Biography of all team members

We are SingaX, a team of PhD students and their advisors from Singapore. Xinyuan Niu and ZhiLiang Chen are PhD students from National Univeristy of Singapore. Vernon Yan Han Toh and Yanchao Li are PhD students from Nanyang Technological University of Singapore. Our supervisors, Zhengyuan Liu, Nancy F. Chen are from Agency for Science, Technology and Research (A*STAR), Singapore.

B Openrouter payload

Below is the payload used to query Openrouter. We used temperature of 0.0, greedy sampling, to minimize randomness in the experiments.

```
payload = {
    "model": "qwen/qwen3-235b-a22b-thinking-2507",
    "temperature": 0.0,
    "provider": {
        "require_parameters": True,
        "quantizations": ["bf16"],
        "allow_fallbacks": False,
        "sort": "price"
    }
}
```

C Prompts to the `Safeguard Generation LLM`

To query the `GPT-5` GitHub Copilot agent, we used the following prompts.

First, we asked the LLM to examine the log file. We attached the system prompt (Python file) and the log file (JSON file) with this prompt.

```
Look through the whole JSON file, and summarize the key
mistakes made by the LLM generation for the tasks.
```

We then asked the LLM to update and improve the prompt based on the points collated from the above query. We attached the system prompt (Python file) with this prompt for the Copilot to modify.

```
Improve the prompts to address the issues
```

Example outputs of these prompts are provided in fig. 2.

D Sample output from the `Safeguard Generation LLM`

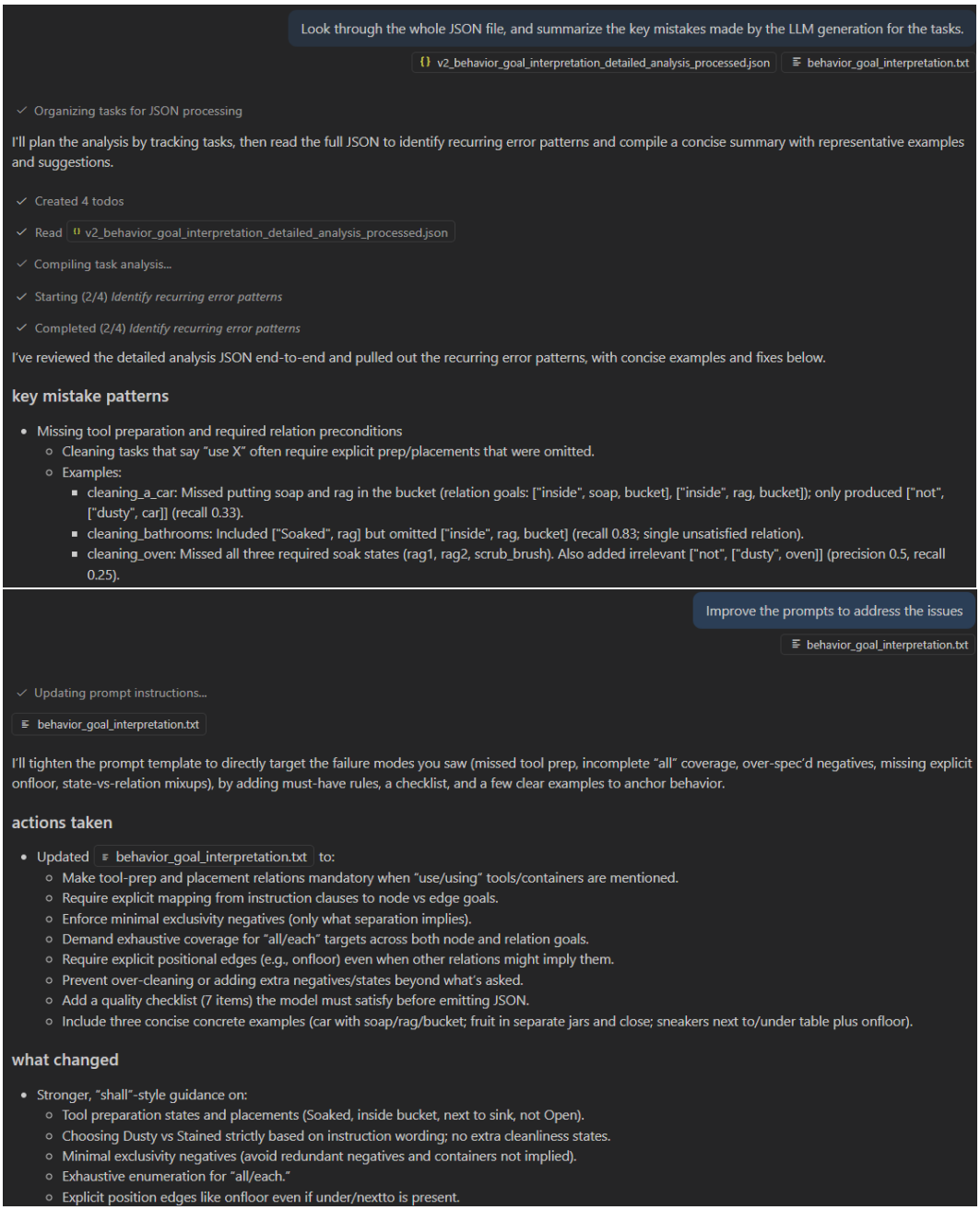

Figure 2: Sample outputs from the `Safeguard Generation LLM` for task `behavior` task `goal_interpretation`. We used Visual Studio's GitHub Copilot extension with `GPT-5` in agent model. Our method can be easily integrated into the coding workflow as the approach is easily implemented directly in the code editor with minimal effort.