Evaluator-Guided LLM Distillation for Embodied Agent Decision-Making

Chinmayan Pradeep Kumar Sreekala[1] and Sanjayan Pradeep Kumar Sreekala[2]

[1] NYU Neuroinformatics Lab
[2]Independent Researcher
cpk286@nyu.edu, san@sankala.me
Both authors contributed equally to this work.

Abstract

We present the winning submission to the NeurIPS 2025 Embodied Agent Interface (EAI) challenge in this report. We treat each module—Goal Interpretation, Subgoal Decomposition, Action Sequencing, and Transition Modeling—as a supervised sequence-to-sequence task, and finetune small Qwen3 models on data constructed using LLM driven evaluator-in-the-loop refinement and the public EAI dataset. On BEHAVIOR, these task-specialized Qwen3 models nearly saturate all four modules and outperform a strong GPT-5-mini baseline; on VirtualHome, they substantially improve performance across the board. We also introduce a learned LLM-as-an-evaluator—a finetuned Qwen3 model that scores and refines candidate outputs—which when combined with retrieval and simple voting yields further gains. Overall, our results show that careful prompt design and evaluator-guided distillation can allow smaller open source models to match or outperform frontier LLMs on embodied reasoning tasks.

1 Introduction

Large language models (LLMs) are increasingly used as high-level controllers for embodied agents and robots: they read natural language instructions, reason about objects and relations, and produce symbolic plans that can be executed in simulation or on real hardware. Most existing evaluations, however, treat the controller as a black box and report only end-to-end task success, which makes it hard to localize whether failures come from misunderstanding the goal, poor decomposition, invalid actions, or incorrect modeling of state changes.

The Embodied Agent Interface (EAI) benchmark [17] addresses this by standardizing an object-centric symbolic interface and decomposing control into four modules: *Goal Interpretation* (GI), *Subgoal Decomposition* (SD), *Action Sequencing* (AS), and *Transition Modeling* (TM). These modules are instantiated in two household simulators, BEHAVIOR [29] and VirtualHome [22], and evaluated with a fixed set of prompts, a public dataset [16], and an evaluator [1]. This setting turns embodied control into a modular, fully supervised benchmark with clear interfaces and metrics.

In this report we study how far small, task-specialized open models can be pushed in this modular EAI setting. Concretely, we treat each module as a sequence-to-sequence task over prompts and outputs, and follow a simple two-stage recipe. First, we use a small zoo of frontier LLMs (including GPT-5-family [19] and Gemini models [6]) to design prompts and probe systematic failure modes. Second, we use these models, the public EAI data, and the official evaluator to construct training sets, and then finetune compact Qwen3 [39] models on these labels. For VirtualHome SD and AS we further add light scaffolding in the form of retrieval-augmented prompting, simple voting, and a *learned LLM-as-evaluator*—a finetuned Qwen3 model trained to score and refine candidate outputs from other models.

2 Related Work

Embodied AI benchmarks and simulators. Interactive simulators for embodied agents include VirtualHome [22], BEHAVIOR [29], AI2-THOR [13], iGibson [28], and Habitat [30] (see also AllenAct [36]). These platforms typically evaluate agents by end-to-end task success on navigation or rearrangement tasks. The Embodied Agent Interface (EAI) benchmark [17] instead standardizes a symbolic, object-centric interface and decomposes control into four modules with shared metrics across BEHAVIOR and VirtualHome.

LLMs for planning and robotics. Large language models have been used as zero-shot or few-shot planners in virtual and real environments [9, 3, 18, 5]. Language Models as Zero-Shot Planners [9] map instructions to action descriptions in VirtualHome; SayCan [3] and Code-as-Policies [18] integrate LLMs with robot skills and executable programs; PaLM-E and related vision–language–action models extend this trend to multimodal control [5]. EAI builds on this line of work by providing a shared symbolic interface and evaluator for modular, rather than purely end-to-end, comparison.

Prompting, distillation, retrieval, and LLM-based evaluators. Prompting techniques such as chain-of-thought and self-consistency [35, 33] improve multi-step reasoning, while synthetic data and distillation frameworks such as Self-Instruct and Distilling Step-by-Step [34, 7] show that small task-specific models can match or surpass larger LLMs when trained on curated pseudo-labels. Retrieval-augmented generation [15] and sentence-level encoders like Sentence-BERT [25] are widely used to ground LLMs in task-specific context, and systems such as DSPy [11] treat multi-stage LLM pipelines as declarative programs that can be optimized automatically. In parallel, LLM-as-a-judge approaches [40] demonstrate that strong LLMs can act as automatic evaluators that correlate well with human preferences.

3 Method

We treat each EAI module as a mapping from a text input (prompt) to a structured output (goals, subgoals, action sequences, or transition-modeling action schemas). Our approach can be broadly grouped into the following strategies:

1. **Prompt tuning:** We run multiple LLMs (GPT-5 family, Gemini 2.5/3.0, Kimi K2-Thinking, Qwen3-30B-A3B-Thinking, etc.) on the official prompts and on custom prompt variants to find strong prompt+model configurations and to characterize typical errors for each module and environment. Due to its great cost–performance tradeoff, we opted to use *GPT-5-mini with medium reasoning effort* as our baseline. Unless specified otherwise, this is the configuration that any 'baseline' refers to in this report. All prompt rewriting was done manually with the help of a library we developed. We experimented with the DSPy [11] framework for automated prompt refinement using evaluator feedback but faced challenges and resource constraints when moving past plain prompt rewriting into more data and cost intensive optimizer methods such as GEPA [2].
2. **Data preparation:**
   Extracted from source. We source training data from the EAI domain assets [17, 16]. When ground-truth labels exist (e.g., BEHAVIOR goals and TM action schemas), we simply attach them to the corresponding prompts, yielding direct supervised input–output pairs.
   LLM driven evaluator-in-the-loop refinement. For tasks without ready-made labels, or where we want to refine noisy labels, we use the official evaluator in the `eai-eval` package [1, 17, 16] to run evaluator-in-the-loop refinement driven by an LLM. The LLM first proposes a candidate output; the evaluator then reports structured errors or scores. This feedback is attached to the original prompt and sent back to the same (or more performant) LLM. Through multiple automated iterations, the model is able to reflect, iterate and improve on the answers to create a significantly stronger training set.
3. **Finetuning:** Supervised finetuning is carried out on Qwen3 models of various sizes, ranging from 0.6B to 32B. For some modules we often replace the long original prompts with compact

templates that remove static instructions, which reduces context length, makes learning easier and makes the resulting models easier to deploy on resource-constrained agents and devices.

4. **Scaffolding:** We also experimented with external scaffolding, such as voting, agents, tool use, and Retrieval-Augmented Generation to supplement inference.

3.1 BEHAVIOR

For BEHAVIOR modules, our strategies primarily consisted of prompt-tuning and model finetuning.

Goal Interpretation (GI). Inspired by the more structured VirtualHome prompts, we found that rewriting the GI instructions into a structured two-shot template (using GPT-5-Pro) improved `gpt-5-mini` performance to an F1 of 0.86. As for finetuning, we construct a training set from gold goals in the EAI dataset [17]. However, these goals are not in the format expected in GI outputs and need transformation; for each task we provide GPT-5-mini with the original GI prompt along with the PDDL goal and ask it to emit the transformed target JSON. Qwen3-0.6B is then finetuned on compact prompts containing only the dynamic fields (relevant objects, initial states, task name, goal instructions; see Appendix E) and the aforementioned outputs. At inference time we applied the same prompt format and were able to achieve a model performance of 0.99 F1.

Subgoal Decomposition (SD). We observed that models perform much better on the VirtualHome SD tasks than on BEHAVIOR, and that the VirtualHome SD prompts are more structured. Motivated by this, we use GPT-5-Pro to rewrite the BEHAVIOR SD prompt into a style that matches VirtualHome. Prompt engineering started from the official SD prompt and `gpt-5-mini`, then added VirtualHome-style formatting and explicit instructions to output one atomic subgoal per line and avoid chaining with "and/or". This yielded a prompt-only baseline with 67.0% task success and 71.0% execution success. To construct finetuning data we run an evaluator-in-the-loop refinement for every SD example: `gpt-5-mini` proposes subgoals, the evaluator scores them, we re-prompt the model with the task, current subgoals, and feedback, and we keep the best of up to four refinements. Qwen3-0.6B is then finetuned on a compact template with task name, relevant objects, initial states, and goal states, and the refined subgoal list as target, achieving 97.0% SD task success.

Action Sequencing (AS). We experimented with several prompt variants in AS and found that explicitly asking the model to re-assert all location predicates after interactions improved performance over baseline. We also saw some improvement when the model is asked to reassert all goal predicates at the end of the task. Outside of prompting, we also experimented with agent scaffolding to implement an LLM-as-a-judge pattern, pairing the inference model against a 'judge' model providing rubric-based feedback on the answer and allowing up to ten agent handoffs during runtime. Another experiment saw a state-management tool introduced that helped the model track the global state of the environment and the objects in it. For training data preparation, we run a refinement loop analogous to SD, seeding plans from the public EAI dataset [32] where available or from the best prompt-only configuration otherwise, and automatically iterating with the evaluator up to eight times. Qwen3-0.6B is finetuned on a compact serialization of initial states, target states, and interactable objects, with the refined plan as target; this achieves 98.0% task success and 100.0% execution success.

Transition Modeling (TM). BEHAVIOR TM asks the model to complete partially specified PDDL action schema (preconditions and effects) [17]. Here we leverage the fact that BEHAVIOR exposes ground-truth domain files [29]: we extract the gold `:precondition` and `:effect` fields for each action and align them with the TM prompts and use this as training data. Prompt-only baselines include several GPT-5-Pro rewrites; the best achieves F1 0.64 and planner success 85.0%. Qwen3-0.6B is finetuned directly on the full TM prompts, with the completed `:action` block as target, and achieves F1 1.00 and planner success 99.0%.

3.2 VirtualHome

For VirtualHome modules we apply the same high-level strategy as for BEHAVIOR—prompt refinement plus finetuning—but with a stronger emphasis on generalization. VirtualHome has many more training tasks and a substantially larger held-out set, so we expect it to benefit more from models that generalize beyond a small number of prompt patterns. Motivated by the strong finetuning gains on the smaller BEHAVIOR tasks, we therefore use larger Qwen3 models [39] (up to Qwen3-32B) and stronger proprietary models such as GPT-5 [19] and Gemini 3.0 [6]. All datasets are also prepared with the evaluator-in-loop approach described in Section 3 unless specified otherwise.

Across a subset of VirtualHome tasks we also tried a specialized embodied model, Gemini Robotics-ER 1.5 [38], but it produced near-zero scores and often emitted outputs that the evaluator could not parse reliably. In contrast, Gemini 3.0 performed strongly and was consistently among the best models we tested on VirtualHome. This may be related to its reported state-of-the-art performance on several multi-modal benchmarks, whereas Robotics-ER appears more narrowly tuned for robotics rather than precise text generation under the EAI interface.

Below we outline the main strategies we employ for VirtualHome modules; module-specific variants are described in the corresponding subsections.

1. **Prompt tuning.** In a few cases, especially during the development phase, we rewrite prompts (e.g., for clarity or formatting) and evaluate their effect on performance.
2. **Supervised finetuning.** For each module we construct the best dataset of labels we can obtain from ground-truth annotations and LLM driven evaluator-in-the-loop refinement (as described in Section 3) and finetune Qwen3 models on this data.
3. **Domain adaptation.** Before module-specific finetuning, we perform an additional finetuning stage on all prompts from both BEHAVIOR and VirtualHome to teach the model the module vocabulary and objectives and to adapt it to the domain. Concretely, we train the model to predict the module name given the full prompt, applying loss over the entire sequence so that it is encouraged to model both the prompt text and the (trivial) label.
4. **Cross-task learning.** We test whether training on multiple VirtualHome modules jointly improves individual module performance.
5. **Cross-dataset learning.** We investigate whether mixing BEHAVIOR and VirtualHome examples during finetuning improves any VirtualHome modules.
6. **Voting.** We combine multiple generations or model variants via simple voting over structured outputs.
7. **Reinforcement learning (RL).** We experiment with reinforcement learning to test whether reasoning models can learn module-specific behavior while retaining chain-of-thought rollouts.
8. **Learned LLM-as-an-evaluator.** We seed the LLM driven evaluator-in-the-loop refinement described in section 3 with multiple model outputs of varying quality and construct a dataset of evaluator feedback, prompts and model outputs that can be used to train an LLM to act as an evaluator.

Goal Interpretation (GI). The original prompts leave the notion of *action goals* somewhat implicit, so our prompt-only baselines focus on clarifying this and on structuring the reasoning process.

For prompting, we use a two-step format: the model is first asked to describe a natural-language solution, and a second pass is then made to restate this solution in the required JSON format. We also explore simple voting schemes: for each task we generate multiple outputs and only accept a goal if at least M out of N outputs agree on it. Sweeping over voting thresholds, we find that a $4/4$ agreement threshold for node and edge goals and a more relaxed $2/4$ threshold for action goals gives the best prompt-only performance.

We then experiment with reinforcement learning, using F1 scores against prepared labels (along with formatting rewards) as the objective. Here we train Qwen3-32B with QLoRA adapters [4] and a GRPO-style policy optimization objective [27] inspired by RLHF [20], but in practice these runs are often unstable and tend to collapse after roughly one epoch. As a result, our final configuration relies on supervised finetuning rather than RL.

For supervised finetuning, we first apply the domain-adaptation stage described in Section 3.2, using prompts from all BEHAVIOR and VirtualHome modules. We then finetune a Qwen3-4B model [39] on a single combined dataset containing GI, SD, and AS examples from both environments to enable cross-task and cross-dataset learning. At inference time, we generate four responses from this finetuned Qwen3-4B model and apply voting with a $2/4$ threshold for node and edge goals and a $1/4$ threshold for action goals.

Subgoal Decomposition (SD). Prompt-only baselines using GPT-5-mini, GPT-5, and Gemini 2.5 Pro achieve high task success (around 88–91%) on the development split, but fail to recreate this in the hidden test set. Finetuned Qwen3 models at various configurations also did not match the performance of the best frontier-models, so in our final pipeline the primary SD generator is a prompt-based Gemini 3.0 model.

To strengthen this prompt-based solution, we augment Gemini 3.0 with semantic example retrieval. We embed task names with the sentence-transformer model `all-MiniLM-L6-v2` [25] and retrieve the five nearest training examples for each test task. These retrieved examples are then included as a 5-shot context in the SD prompt, yielding a simple retrieval-augmented (RAG) setup for subgoal decomposition [15].

During the development phase we also ran several models through the LLM driven evaluator-in-the-loop refinement process detailed previously in Section 3, wherein the model iteratively proposes subgoal lists, receives feedback from the EAI evaluator, and then updates its outputs. We realized that the more difficult tasks were naturally overrepresented in this set of iterations—as entries that were more resistant to change would go through multiple loops before (occasionally) resolving into a successful output. By seeding this system with outputs from models of various sizes, and then pairing the inputs, outputs and the actionable feedback from the evaluator at each step, we were able to construct a training set that could be used to finetune a Qwen3-32B model to act as a **learned LLM-as-an-evaluator** for SD tasks. This model was then used to give feedback on our best performing output at the time (Gemini 3.0 with retrieval) and contributed to an increase of 1.5% in task success rate.

On further analysis of the training data, we noticed that the evaluator marked 1,074 cases as correct and 1,133 cases as incorrect; so "requires fixing" is slightly more reinforced than "no change necessary" ($\sim$51% vs. 49%). To counter this sensitivity, we introduced an element of cross-model confidence in the following manner: within the set of entries that the evaluator model deems as requiring correction, we only continue with refinement for outputs that do not match our best performing finetuned model, which was a Qwen3-32B model trained with a cross-module configuration. In other words, if both models agree on an output that was flagged as wrong, we consider it as incorrectly flagged by the evaluator model. This final refinement resulted in a task success rate of 78.7%.

Action Sequencing (AS). For VirtualHome AS we map the task description, goal specification, and scene context to a full action script, which is then executed in the simulator and scored by the evaluator. We compared multiple backbone models, including Gemini 3.0, GPT-5, Qwen3-30B-A3B-Thinking in the development phase. For training data construction, we use the LLM driven evaluator-in-the-loop procedure seen in BEHAVIOR AS. We explored a large set of data mixes for finetuning (see Appendix B), combining the strategies from Section 3: standard supervised finetuning on AS labels, domain adaptation, cross-task learning over GI/SD/AS, and cross-dataset learning that mixes BEHAVIOR and VirtualHome examples. (See Section 5.)

The most effective student in our experiments is a Qwen3-32B model trained in two stages. First, we apply the domain-adaptation stage over all BEHAVIOR and VirtualHome prompts, as described in Section 3.2. Cross-task training (GI/SD/AS) and cross-dataset mixing with BEHAVIOR gave competitive but slightly worse AS performance. The domain-adapted Qwen3-32B finetuned only on VirtualHome AS yields the final VirtualHome AS score of 82.6% task success on the leaderboard.

Transition Modeling (TM). Here we follow a similar approach to BEHAVIOR since training data closely matches the evaluation distribution. Motivated by our findings that simpler prompts work better in the BEHAVIOR environment, we use an even more streamlined setup. Starting from the gold PDDL action files, we extract a supervised dataset that maps each action name to its fully instantiated parameters, preconditions, and effects. We observe that there are only 33 actions in the dataset, and 17 distinct actions across 1,500 prompts in the test set, so the effective learning burden on the model is quite small. Hence we opt for a small model for finetuning and train a Qwen3-0.6B model on this mapping task. At evaluation time, we extract the actions to be executed from the prompt, run the TM model on each action, and merge the predicted action schemas back into the final answer.

4 Implementation Details

Model zoo. We experimented with a mixture of open and closed-source large language models (LLMs) throughout the challenge. On the open side, we used several Qwen variants: a 32B parameter model (`Qwen3-32B`), smaller models at 0.6B, 4B, and 8B parameters, and the reasoning-focused `Qwen/Qwen3-30B-A3B-Thinking-2507`. As for APIs, we used Kimi K2 ("thinking" variant) [12], multiple generations of Gemini (Gemini 2.5, Gemini 2.5 Pro, and Gemini 3.0), and the GPT-5 family (GPT-5-nano, GPT-5-mini, GPT-5, and GPT-5-Pro). GPT-5-mini served as our main baseline, GPT-5-Pro (used through chat.openai.com) for prompt rewriting, and GPT-5-nano to seed data for the evaluator model. Gemini executions were sometimes unreliable and were therefore set up to use GPT-5-mini as a fallback.

Finetuning and inference setup. We enable `completion_only_loss` so that only the target span (goal JSON, subgoal list, action sequence, or PDDL schema) contributes to the loss, with the exception of training on the prompts during domain adaptation learning. All SFT runs use TRL's `SFTTrainer` [10]. Unless specified otherwise, we used a random seed of 42, and AdamW optimizer. For inference, we used near-greedy decoding (temperature 0). All other parameters were set to their defaults. When we required output diversity (typically for voting-based pipelines) we set the temperature to 1. We did not perform hyperparameter sweeps and instead reused a single stable configuration across all runs. In total we estimate a $2000 spend on compute and API.

- **Rental cloud GPUs (RunPod).** We ran most small and mid-sized experiments on rental cloud GPUs via RunPod [26]. For Qwen3 0.6B, unless otherwise stated, we used a context length of $8{,}192$ tokens, learning rate 1×10^{-4}, AdamW optimizer in 8-bit, an effective global batch size of ≈ 4 (via gradient accumulation), a cosine learning-rate schedule without warmup, and 75–100 epochs depending on the module. For mid-size models, i.e. Qwen3 4B and Qwen3 8B, we configured the run to use 5-10 epochs. These models are full-weight finetuned.
  For reinforcement learning we utilized a $2\times$A100 setup with vLLM and a custom GRPO script running for 1–2 epochs—but found it to be too cost intensive for any meaningful iteration.
- **High-performance computing.** For the larger Qwen3-32B models, we opted to use LoRA [8] finetuning. We finetune these models on the NYU Greene HPC cluster using DeepSpeed [24, 23] ZeRO-2 for data-parallel training across two H100 GPUs. We also experimented with DeepSpeed ZeRO-1 and ZeRO-3 and found ZeRO-2 to reliably maximize GPU utilization. Epochs range from 2 to 30, with 20 being standard. A run of 20 epochs over ~ 350 training samples (which is typical for our datasets) takes roughly 1.5–2 hours in our environment. In total, we used approximately 257 hours of H100 wall time across all experiments. For inference, we created a vLLM script [31, 14] that can either consume merged checkpoints or attach LoRA adapters, with tokenizer + prompts built via the Qwen chat template. For these jobs we keep tensor_parallel_size=1, so the entire 32B model sits on a single H100 while vLLM handles batching/scheduling and targeted 90% GPU utilization. More information on parameters is detailed in Appendix C.

Tooling Our codebase is implemented in Python using PyTorch [21], HuggingFace Transformers [37], TRL [10], DeepSpeed [24, 23], vLLM [31, 14], sentence-transformers [25], and the official EAI evaluator [16], plus custom libraries we created for prompt extraction, batched API calls, and evaluator-guided refinement. More information about these projects is detailed in Appendix D.

5 Evaluation Results

Tables 1 and 2 report official evaluation scores on the BEHAVIOR and VirtualHome environments, respectively. We compare a strong baseline model (`gpt-5-mini-medium`) against our module-specialized submissions. For BEHAVIOR, our final system uses separate finetuned Qwen3-0.6B models for each module. For VirtualHome, we employ a mixture of larger Qwen3 variants and a Gemini 3-based evaluator, choosing the best configuration per module. Columns correspond to the four challenge modules; higher is better (↑). "GI" = Goal Interpretation, "SD" = Subgoal Decomposition, "AS" = Action Sequencing, and "TM" = Transition Modeling. GI is evaluated with predicate F1 and reported as F1×100, SD and AS are evaluated with task success rate (%), and TM is evaluated with $(\mathrm{F1} \times 100 + \text{planner success})/2$. A value of "–" indicates that a given model is not used or evaluated for that module. Best scores for each module in each environment are in bold.

Table 1: BEHAVIOR: official evaluation scores for the `gpt-5-mini-medium` baseline and our module-specialized Qwen3-0.6B models (higher is better).

Model	GI F1 (×100)↑	SD Succ.%↑	AS Succ.%↑	TM Avg.↑
Baseline (`gpt-5-mini-medium`)	78.6	50.0	68.0	80.0
GI model (Qwen3-0.6B finetuned, ours)	**99.6**	–	–	–
SD model (Qwen3-0.6B finetuned, ours)	–	**97.0**	–	–
AS model (Qwen3-0.6B finetuned, ours)	–	–	**98.0**	–
TM model (Qwen3-0.6B finetuned, ours)	–	–	–	**99.5**

Table 2: VirtualHome: official evaluation scores for the `gpt-5-mini-medium` baseline and our module-specific models and techniques (higher is better).

Model	GI F1 (×100)↑	SD Succ.%↑	AS Succ.%↑	TM Avg.↑
Baseline (`gpt-5-mini-medium`)	34.6	71.1	71.0	58.2
GI model (Qwen3-4B finetuned + voting, ours)	**65.4**	–	–	–
SD model (Gemini 3 + finetuned LLM evaluator, ours)	–	**78.7**	–	–
AS model (Qwen3-32B finetuned, ours)	–	–	**82.6**	–
TM model (Qwen3-0.6B finetuned, ours)	–	–	–	**99.9**

Extended results for other models and techniques are provided in Appendix B.

6 Analysis and Discussion

6.1 Error Analysis

Across modules we observe two dominant families of errors.

Human–simulator mismatch. When prompted naively, frontier LLMs often produce "human-looking" answers that are short, efficient, and intuitively correct, but fail under the EAI evaluator because they omit non-obvious states required by the simulator. Typical examples include failing to track that objects change when items are sliced (e.g., using the pre-slice ID in later actions), or skipping intermediate steps (e.g. `OPEN`/`CLOSE`) that appear redundant to a human reader but are mandatory in the underlying domain. We suspect that humans, if forced to interact purely through this symbolic interface, would also underperform on many of these tasks. This motivates prompts that explicitly have instructions such as "your objective is not to find the most efficient answer but rather one that demonstrates that all expected goals are completed to the simulator," and motivates finetuning models to internalize these hidden conventions rather than relying solely on natural-language hints.

Symbolic bookkeeping and formatting. A second class of failures arises from purely symbolic or formatting issues: missing or duplicated predicates, incorrect argument ordering, or subtle violations of the expected output structure. In BEHAVIOR SD, chaining complex formulas with `and`/`or` in a single line often leads to invalid or under-specified subgoals. In VirtualHome AS, the original JSON-like output suggestion encourages models to invent artificial keys such as `"WALK-2"` for repeated actions; thinking-style models appear to waste capacity maintaining these keys instead of focusing on action correctness. Relaxing such constraints (e.g., treating the plan as an ordered list rather than a JSON object with unique keys) yields small but consistent improvements (e.g., $70.2 \rightarrow 71.8$ task success in VirtualHome AS).

6.2 Cross-Environment Comparison

Compared to BEHAVIOR, VirtualHome offers many more tasks with richer semantics, and a substantially larger held-out set. Prompt structure is more regular (headings, tables, explicit rules), which partly explains why prompt-only baselines perform well on the development phase prompts, especially for SD. However, the diversity of the test distribution makes GI and SD substantially harder: finetuning and domain-wide pretraining help, but do not fully close the gap, and hybrid systems that combine strong API models (Gemini 3.0), retrieval, voting, and a learned evaluator remain competitive. Cross-dataset finetuning (mixing BEHAVIOR and VirtualHome) modestly improves VirtualHome GI scores, but can hurt AS and SD if pushed too far, suggesting that the two environments encode slightly different conventions about how goals and decompositions should be written.

6.3 Insights and Lessons Learned

Three insights stood out:

- **Prompt structure is part of the system.** The gap between the original BEHAVIOR prompts and our rewritten VirtualHome-style prompts shows that interface design alone can move performance by tens of points without changing the underlying model or data. Structured prompts with clear headings, tables, and explicit output rules make both prompt-based baselines and finetuned models substantially easier to optimize.
- **Finetuning internalizes the evaluator.** Supervised finetuning on evaluator-verified data allows small models to absorb the evaluator's implicit criteria: which states must be made explicit, when redundancy is preferred, and how to resolve ambiguities in ways that maximize the score. This helps explain why 0.6B–4B students can outperform frontier models on several modules.
- **Scaffolding helps less than better data.** Agent-style scaffolding (LLM-as-a-judge, state-management tools), retrieval-augmented prompting, prompt tuning, and GRPO-based RL all provided some benefit in pilot experiments, but were ultimately dominated by straightforward supervised finetuning once high-quality labels were available. Under our compute budget, investing effort into cleaner prompts and evaluator-guided data construction yielded larger gains than investing in more complex inference-time or training-time scaffolds.

7 Conclusion

We presented a unified approach to the Embodied Agent Interface (EAI) challenge based on prompt refinement, evaluator-guided data construction, and supervised finetuning of small open models. On BEHAVIOR, this pipeline allows a 0.6B Qwen3 model to achieve very strong performance across all four modules. On VirtualHome, which has a larger and more diverse task set, we obtain strong but non-saturated results by combining finetuned Qwen3 models with frontier API models, retrieval, voting, and a learned LLM-as-evaluator.

Overall, we find that careful control over prompts, output formats, and evaluator-verified labels matters at least as much as model size: once distilled on evaluator-guided data, small Qwen3 models can match or outperform frontier models on several modules. At the same time, VirtualHome GI, SD, and AS remain far from saturated, especially on longer and more loosely specified tasks, suggesting room for future work on more realistic, partially observed settings and on transferring the same recipe to end-to-end embodied agents beyond EAI.

Reproducibility Statement

We build directly on the EAI starter kit, evaluator [16] and the public EAI dataset [32]. All models are standard Qwen3 variants [39] or proprietary models documented by their providers. Prompt templates are provided in Appendix E, and key hyperparameters are listed in Section 4. Code repositories and finetuned checkpoints detailed in Appendix D will be made public after the competition end.

Acknowledgments

We extend our gratitude to the NYU Neuroinformatics Lab for providing access to resources on the NYU Greene HPC supercomputer cluster. We also acknowledge the use of AI tools, including GPT-5.1, Gemini 3.0, and Claude 4.5, which assisted in writing this report and in code development.

References

[1] Eai evaluation toolkit: Getting started. https://embodied-agent-eval.readthedocs.io/en/latest/get_started.html. Accessed: 30 November 2025.

[2] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[3] Michael Ahn, Anthony Brohan, Noah Brown, et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, 2022.

[4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. *arXiv preprint arXiv:2305.14314*, 2023.

[5] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, et al. Palm-e: An embodied multimodal language model. In *International Conference on Machine Learning*, 2023.

[6] Google DeepMind. A new era of intelligence with gemini 3. https://blog.google/products/gemini/gemini-3/, 2025.

[7] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[8] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[9] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, 2022.

[10] Hugging Face. TRL: Transformer reinforcement learning library. https://github.com/huggingface/trl, 2023.

[11] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[12] Kimi Team. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[13] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, Aniruddha Kembhavi, Abhinav Gupta, and Ali Farhadi. Ai2-thor: An interactive 3d environment for visual ai. In *arXiv preprint arXiv:1712.05474*, 2017.

[14] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

[16] Manling Li, Shiyu Zhao, Qineng Wang, et al. Embodied agent interface: Benchmarking LLMs for embodied decision making (code and evaluator). `https://github.com/embodied-agent-interface/embodied-agent-interface`, 2024.

[17] 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.

[18] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *Conference on Robot Learning*, 2022.

[19] OpenAI. GPT-5 system card. `https://cdn.openai.com/gpt-5-system-card.pdf`, 2025.

[20] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[22] Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[23] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[24] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations for training deep learning models at scale. *arXiv preprint arXiv:2007.00324*, 2020.

[25] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019.

[26] RunPod. Runpod GPU cloud. `https://www.runpod.io/`, 2025. Accessed: 2025-11-30.

[27] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[28] Bokui Shen, Fei Xia, Chengshu Li, Roberto Martín-Martín, Linxi Fan, Guanzhi Wang, Claudia Pérez-D'Arpino, Shyamal Buch, Sanjana Srivastava, Lyne P. Tchapmi, Micael E. Tchapmi, Kent Vainio, Josiah Wong, Li Fei-Fei, and Silvio Savarese. igibson 1.0: A simulation environment for interactive tasks in large realistic scenes. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021.

[29] Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, C. Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Proceedings of the 5th Conference on Robot Learning*, 2022.

[30] Andrew Szot, Alex Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. *arXiv preprint arXiv:2106.14405*, 2021.

[31] vLLM contributors. vLLM: Easy, fast, and cheap LLM serving. `https://github.com/vllm-project/vllm`, 2023.

[32] Qineng Wang, Manling Li, et al. Embodiedagentinterface dataset. `https://huggingface.co/datasets/Inevitablevalor/EmbodiedAgentInterface`, 2024.

[33] Xuezhi Wang, Jason Wei, Dale Schuurmans, et al. Self-consistency improves chain-of-thought reasoning in language models. *Transactions on Machine Learning Research*, 2023.

[34] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2022.

[35] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

[36] Luca Weihs, Jordi Salvador, Klemen Kotar, Unnat Jain, Kuo-Hao Zeng, Roozbeh Mottaghi, and Aniruddha Kembhavi. Allenact: A framework for embodied ai research. *arXiv preprint arXiv:2008.12760*, 2020.

[37] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, 2020.

[38] Fei Xia, Kendra Byrne, et al. Gemini robotics: Bringing ai into the physical world. *arXiv preprint arXiv:2503.20020*, 2025.

[39] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[40] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

A Biography of all team members

Team name: AxisTilted2

Chinmayan Pradeep Kumar Sreekala is a Graduate Research Assistant at NYU Neuroinformatics Lab and a recent CS MS graduate from NYU.

Sanjayan Pradeep Kumar Sreekala is an applied research scientist in industry, working on natural language processing and generative models.

B Extended Results

B.1 BEHAVIOR

B.1.1 Goal Interpretation (B-GI)

Model	F1 score (%)↑
Base models (default prompts)	
GPT-5-mini (baseline)	78.6
Kimi-K2-Thinking	80.0
GPT-5	82.3
Qwen3-30B-A3B-Thinking-2507	82.5
Gemini 2.5 Pro	84.4
Prompt-engineered variant	
Baseline w VirtualHome-style prompt	86.1
Finetuned model (ours)	
B-GI finetune (Qwen3-0.6B)	**99.6**

Table 3: BEHAVIOR Goal Interpretation (B-GI) F1 scores for base models, a prompt-engineered variant, and our finetuned model (combining development and evaluation results).

B.1.2 Subgoal Decomposition (B-SD)

Model / technique	Task success (%)↑
Base models (default prompts)	
GPT-5-mini (baseline)	50.0
Kimi-K2-Thinking	42.0
GPT-5	50.0
Gemini 2.5 Pro	53.0
Prompt-engineered variants (on GPT-5-mini)	
Avoid chaining goals (and/or)	55.0
VirtualHome-style prompt	65.0
VirtualHome-style prompt + avoid chaining goals	67.0
Finetuned model (ours)	
B-SD finetune (Qwen3-0.6B)	**97.0**

Table 4: BEHAVIOR Subgoal Decomposition (B-SD) task success rates for base models, prompt-engineered variants, and our finetuned model (combining development and evaluation results). The prompt-engineered variants are all applied on top of GPT-5-mini. The "Avoid chaining goals (and/or)" variant instructs the model to avoid chaining multiple subgoals with conjunctions where unnecessary, which reduces ambiguity and overly long decompositions. The "VirtualHome-style prompt" variant adapts a cleaner VirtualHome markdown-style prompt to BEHAVIOR. The combined variant ("VirtualHome-style prompt + avoid chaining goals") uses both modifications and yields the best prompt-only result.

B.1.3 Action Sequencing (B-AS)

Model / technique	Task success (%)↑	Execution success (%)↑
Base models (default prompts)		
GPT-5-mini (baseline)	68.0	76.0
GPT-5	65.0	69.0
Gemini 2.5 Pro	57.0	66.0
Kimi-K2-Thinking	74.0	84.3
Prompt-engineered variants (on GPT-5-mini)		
Soft Location Assertion	77.0	85.0
Strong Location Assertion	79.0	85.0
Generalized Assertion	71.0	79.0
Strong Location assertion + LLM-as-a-judge	77.0	84.0
Base prompt + state-management tool	75.0	86.5
Finetuned model (ours)		
B-AS finetune (Qwen3-0.6B)	**98.0**	**100.0**

Table 5: BEHAVIOR Action Sequencing (B-AS) task and execution success rates for base models, prompt-engineered variants, and our finetuned model. The prompt-engineered variants are all applied on top of GPT-5-mini. The "Soft Location Assertion" and "Strong Location Assertion" variants guide the model to re-assert location predicates after relevant interactions. The "Generalized assertion" variant extends this pattern beyond locations to all subgoal predicates. The "Strong Location assertion + LLM-as-a-judge" variant additionally uses a secondary LLM to judge the output. Finally, the "state-management tool" variant augments the base prompt with an explicit external state-tracking tool.

B.1.4 Transition Modeling (B-TM)

Model / technique	Planner success (%)↑	F1 (%)↑
Base models (default prompts)		
GPT-5-mini (baseline, dev leaderboard)	80.0	59.5
GPT-5-mini (baseline, eval)	83.0	61.1
GPT-5	94.3	36.5
Kimi-K2-Thinking	92.0	61.3
Gemini 2.5 Pro	68.0	58.4
Prompt-engineered variants (on GPT-5-mini)		
Simplified prompt (no example)	89.0	25.0
Simplified prompt (with example)	75.0	25.4
Duplicate formatting instructions removed	81.0	57.8
Prompt rewritten by GPT-5 (cleaner prompt)	85.0	64.1
Finetuned model (ours)		
B-TM finetune (Qwen3-0.6B)	**99.0**	**100.0**

Table 6: BEHAVIOR Transition Modeling (B-TM) planner success and F1 scores for base models, prompt-engineered variants, and our finetuned model. The prompt-engineered variants are all applied on top of GPT-5-mini. The "simplified prompt" variants strip down and simplify the original B-TM prompt, either without or with a single in-context example, while the "duplicate formatting instructions removed" variant removes redundant formatting directives. The "prompt rewritten by GPT-5" variant uses GPT-5 to rewrite the original prompt for clarity and conciseness. Our finetuned Qwen3-0.6B B-TM model substantially outperforms all prompt-only configurations on both planner success and F1.

B.2 VirtualHome

B.2.1 Goal Interpretation (development phase)

Model / technique	F1 score (%)↑
Base models (single-sample, default prompts)	
GPT-5-mini (baseline)	42.9
GPT-5	48.7
GPT-5 Pro (single-prompt rewrite)	46.2
Gemini 2.5 Pro	44.5
Prompt variants on GPT-5-mini (single-sample)	
Two-step prompt (NL reasoning → JSON goals)	48.9
Two-step prompt + action-goal hints	50.9
Voting ensembles on GPT-5-mini	
Baseline (single-prompt), vote 4/4	45.6
Two-step + action-goal hints, vote 4/4	52.0
Two-step + action-goal hints, vote 3/4	52.8
Two-step + action-goal hints, vote 2/4	53.8
Two-step + action-goal hints, vote 1/4	52.0
Two-step + action-goal hints, vote node 4/4, rest 2/4	**55.3**

Table 7: VirtualHome Goal Interpretation (GI) development-phase F1 scores for base models, prompt variants, and voting ensembles. All prompt and voting variants are applied on top of GPT-5-mini. The *two-step prompt* first asks the model to produce a natural-language solution and only then to restate it in the required JSON goal format. The "two-step + action-goal hints" variant further clarifies in the instructions that high-level action goals are distinct from primitive actions. The voting ensembles explore different majority thresholds over 4 sampled outputs, including a mixed strategy with stricter voting for node predictions.

Model / configuration	F1 score (%)↑
Baseline	
GPT-5-mini (prompt-only baseline)	34.6
Qwen3-4B supervised finetuning	
Qwen3-4B (GI+SD+AS mix, no domain adapt)	64.0
Qwen3-4B (GI-only, no domain adapt)	62.1
Qwen3-4B (V+B mix, no domain adapt)	64.0
Qwen3-4B (domain adapt 1 epoch → V+B mix)	64.3
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs)	65.2
Qwen3-4B (domain adapt 4 epochs → full-domain V+B mix)	64.7
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs, vote 1/4)	64.8
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs, vote 2/4)	65.4
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs, vote 3/4 node, 3/4 edge, 2/4 action)	65.4
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs, vote 2/4 node+edge, 1/4 action)	**65.4**
Qwen3-8B supervised finetuning	
Qwen3-8B (V+B mix, no domain adapt)	64.0
Qwen3-32B supervised finetuning	
Qwen3-32B (GI-only, no domain adapt)	61.6
Qwen3-32B (GI-only, 4-way voting, no domain adapt)	60.8
Qwen3-32B (domain adapt 1 epoch → VGI finetune)	62.1
Qwen3-32B (GI+SD+AS mix, 10 epochs, no domain adapt)	60.4
Qwen3-32B (domain adapt 1 epoch → GI+SD+AS mix, 10 epochs)	61.2
Qwen3-32B (domain adapt 5 epochs → V+B mix, 10 epochs)	64.2

Table 8: VirtualHome Goal Interpretation (GI) evaluation-phase F1 scores for GPT-5-mini and finetuned Qwen3 models. "GI-only" and "GI+SD+AS mix" denote single-task versus multi-task training across GI, SD, and AS examples. "V+B mix" indicates cross-dataset finetuning that mixes VirtualHome and BEHAVIOR data. The notation "domain adapt k epochs → X, N epochs" denotes a two-stage procedure where we first run the domain-adaptation stage for k epochs over all BEHAVIOR and VirtualHome prompts (Section 3.2), and then finetune on objective X for N epochs (or with voting, when specified). The voting configurations ensemble 4 generations with different majority thresholds across node, edge, and action goals. Our best configuration is the Qwen3-4B model with 5 epochs of domain adaptation, followed by 10 epochs of V+B mix finetuning and asymmetric voting (2/4 for node and edge, 1/4 for action goals).

B.2.2 Subgoal Decomposition

B.2.3 Subgoal Decomposition (development phase)

Model / technique	Task success (%)↑	Execution success (%)↑
GPT-5-mini (baseline)	90.5	93.2
GPT-5	**91.4**	**93.8**
Gemini 2.5 Pro	88.5	92.0

Table 9: VirtualHome Subgoal Decomposition (VSD) development-phase task and execution success rates for different base models under a default prompt. GPT-5 achieves the best development performance among the models evaluated.

B.2.4 Subgoal Decomposition (evaluation phase)

Model / configuration	Task success (%)↑	Execution success (%)↑
Prompt-only and simple baselines		
GPT-5-mini (prompt-only baseline)	71.1	86.3
Gemini 3	74.5	90.3
Finetuned Qwen3 models (no retrieval)		
Qwen3-32B (SD-only finetune)	70.8	88.6
Qwen3-32B (GI+SD+AS mix, 10 epochs)	71.0	88.7
Qwen3-32B (domain adapt 1 epoch $\rightarrow$ GI+SD+AS mix, 10 epochs)	69.3	87.9
Qwen3-4B (V+B mix, no domain adapt)	70.2	87.5
Qwen3-4B (domain adapt 2 epochs $\rightarrow$ V+B mix)	68.1	88.9
Qwen3-4B (domain adapt 5 epochs $\rightarrow$ V+B mix)	64.7	**91.3**
Qwen3-4B (domain adapt 4 epochs $\rightarrow$ V+B mix)	60.2	90.5
Retrieval and few-shot prompting (RAG)		
GPT-5-mini, 5-shot with RAG	75.5	88.6
Gemini 3, 5-shot with RAG	77.6	91.1
Gemini 3, 10-shot with RAG	77.0	91.1
Model-as-an-evaluator and final configuration		
Model-as-an-evaluator (Gemini 3 + Qwen3-32B evaluator)	78.5	91.2
Model-as-an-evaluator + confidence filter (Qwen3-32B GI+SD+AS)	**78.7**	91.2

Table 10: VirtualHome Subgoal Decomposition (VSD) evaluation-phase task and execution success rates for prompt-only baselines, finetuned Qwen3 models, retrieval-based few-shot prompting, and our model-as-an-evaluator setup. "GI+SD+AS mix" denotes multi-task training across the three modules, while "V+B mix" indicates cross-dataset finetuning that mixes VirtualHome and BEHAVIOR data. The notation "domain adapt k epochs $\rightarrow$ X" denotes a two-stage procedure where we first run the domain-adaptation stage for k epochs over all BEHAVIOR and VirtualHome prompts (Section 3.2), and then finetune on objective X. The 5-shot and 10-shot RAG variants use a small number of retrieved training examples as in-context demonstrations. The "Model-as-an-evaluator" configuration trains a Qwen3-32B model to act as an evaluator over candidate decompositions, and the final "model-as-an-evaluator + confidence filter" configuration further combines evaluator feedback with a cross-model confidence check against our best finetuned SD model, yielding the best overall task success of 78.7%.

B.2.5 Action Sequencing

B.2.6 Action Sequencing (development phase)

Model / technique	Task success (%)↑	Execution success (%)↑
Base models (default prompts)		
Baseline (competition prompt)	70.2	75.7
GPT-5-mini (local)	72.1	75.7
GPT-5	**72.6**	77.6
Qwen3-30B-A3B-Thinking-2507	62.6	70.8
Prompt variant on GPT-5		
Valid-format	71.8	**78.4**

Table 11: VirtualHome Action Sequencing (V-AS) development-phase task and execution success rates for different backbone models and a simple prompt variant. The "valid-format" configuration uses GPT-5 with instructions to use a valid-JSON format

B.2.7 Action Sequencing (evaluation phase)

Model / configuration	Task success (%)↑	Execution success (%)↑
Qwen3-32B: AS-only finetuning		
Qwen3-32B (AS, 20 epochs)	80.7	95.2
Qwen3-32B (domain adapt 1 epoch → AS, 20 epochs)	81.7	96.0
Qwen3-32B (domain adapt 2 epochs → AS, 20 epochs)	**82.6**	94.3
Qwen3-32B (domain adapt 2 epochs → AS, 30 epochs)	79.2	96.0
Qwen3-32B (domain adapt 5 epochs → AS, 20 epochs)	81.9	93.9
Qwen3-32B: cross-task / cross-dataset finetuning		
Qwen3-32B (GI+SD+AS mix, 10 epochs)	81.1	95.0
Qwen3-32B (domain adapt 1 epoch → GI+SD+AS mix, 10 epochs)	81.1	94.1
Qwen3-32B (domain adapt 2 epochs → GI+SD+AS mix, 10 epochs)	77.3	92.2
Qwen3-32B (domain adapt 2 epochs → V+B mix, 10 epochs)	79.6	91.6
Qwen3-4B: cross-dataset V+B mix		
Qwen3-4B (V+B mix, no domain adapt)	75.8	93.5
Qwen3-4B (domain adapt 1 epoch → V+B mix, 10 epochs)	80.3	97.1
Qwen3-4B (domain adapt 5 epochs → V+B mix, 10 epochs)	78.6	**97.7**
Qwen3-4B (domain adapt 4 epochs → V+B mix, 10 epochs)	73.1	93.1

Table 12: VirtualHome Action Sequencing (V-AS) evaluation-phase task and execution success rates for Qwen3-32B and Qwen3-4B models under different finetuning regimes. "AS" denotes training only on VirtualHome AS labels. "GI+SD+AS mix" denotes cross-task training across goal interpretation, subgoal decomposition, and action sequencing. "V+B mix" indicates cross-dataset finetuning that mixes VirtualHome and BEHAVIOR AS examples. The notation "domain adapt k epochs → X, N epochs" indicates a two-stage procedure where we first run the domain-adaptation stage for k epochs over all BEHAVIOR and VirtualHome prompts (Section 3.2), and then finetune on objective X for N epochs. Our best V-AS configuration is the domain-adapted Qwen3-32B trained only on VirtualHome AS (task success 82.6%), while Qwen3-4B with V+B mix and domain adaptation achieves the highest execution success (97.7%).

B.2.8 Transition Modeling

Model / technique	Planner success (%)↑	F1 (%)↑
Base models (default prompts)		
GPT-5-mini (baseline)	84.8	43.1
GPT-5	88.4	32.3
Gemini 2.5 Pro	59.5	**48.5**
Prompt variant on GPT-5-mini		
GPT-5-mini (prompt rewritten by GPT-5)	**95.3**	43.1

Table 13: VirtualHome Transition Modeling (TM) development-phase planner success and F1 scores for base models and a simple prompt variant. Rewriting the GPT-5-mini baseline prompt with GPT-5 substantially improves planner success, while Gemini 2.5 Pro achieves the best F1 among the base models.

Model / configuration	Planner success (%)↑	F1 (%)↑
Qwen3-0.6B (TM finetuned, ours)	**99.8**	**100.0**

Table 14: VirtualHome Transition Modeling (TM) evaluation-phase results. Our finetuned Qwen3-0.6B TM model, trained on action-level schemas as described in Section 3.2, achieves near-perfect planner success and F1 on the official evaluation split.

Our ablation strategy was to keep experiments moderately resource-constrained. As a result, most combinations of model configurations, sizes, and datasets were chosen not to exhaustively cover the design space, but rather to maximize performance and insight under a limited number of carefully selected settings.

C Additional Hyperparameter Details

Table 15: Training and inference hyperparameters for HPC runs.

Parameter	**Values used**
Inference	
`temperature`	0
`top_p`	0.9
`max_new_tokens`	2048
`max_model_len`	16,384
Training	
`num_train_epochs`	2 to 30
`per_device_batch`	1
`grad_accum`	8
`learning_rate`	2×10^{-4}
`max_seq_length`	4,096
`LoRA r/alpha/dropout`	4 / 128 / 0.05

D Code Repositories and Model Checkpoints

Table 16: GitHub repositories associated with this work.

Repository	**Function**
Custom libraries	
`CinnamonRolls1/eai-extractor`	Prompt tuning, dynamic and static extraction, template prettification, and more
`spsanps/gpt-api-wrapper`	Lightweight wrapper that allows multi-threaded conversation and other inference adjacent features on top of API models
Centralized repositories	
`spsanps/eai-experiments`	Repository exploring a breadth of experiments; includes prompt and dataset tuning, finetuning and inference for rental cloud GPUs and more
`CinnamonRolls1/inference-central`	Unified repository for agentic inference, tool scaffolding for API inference, as well as HPC sbatch files, training and inference scripts and outputs
Narrow repositories	
`CinnamonRolls1/docker-eval`	Wrapper around the eai-eval package for local evaluation; home of the initial DSPy efforts and local evaluation of results
`CinnamonRolls1/gt-miner`	Freeform dataset exploration and curation, primarily features evaluator feedback refinement loop via masked batch conversations

Model checkpoints are available at `https://huggingface.co/AxisTilted2`. Private repositories will be made public after competition end.

E Prompt Templates

All prompt templates are used along with the chat-template of the corresponding model.

E.1 BEHAVIOR

This appendix lists the compact prompt formats we use for fine-tuning Qwen3-0.6B on the BEHAVIOR modules. These templates are used for constructing SFT prompt examples and at inference time for the student. These are the prompt formats and templates used for best and/or final submission to the contest.

BEHAVIOR Goal Interpretation (GI). For GI we use:

```
{relevant_objects}
---
{inital_states}
---
{task_name}
{goal_description}
```

BEHAVIOR Subgoal Decomposition (SD). For SD we use:

```
{task_name}
---
{relevant_objects}
---
{initial_states}
---
{goal_states}
```

BEHAVIOR Action Sequencing (AS). For AS we use:

```
{initial_states}
---
{target_states}
---
{interactable_objects}
```

BEHAVIOR Transition Modeling (TM). For TM we do not introduce a separate compact template. We fine-tune directly on the full official TM prompt, with the completed `:action` blocks from the ground-truth domain file as the target span.

E.2 VirtualHome

This appendix lists the compact prompt formats we use for VirtualHome modules. It represents the prompt formats and templates used for the best and/or final submission to the contest.

E.2.1 Goal Interpretation

No custom prompts used for the final submission.

E.2.2 Subgoal Decomposition

For the final submission we use a K-shot prompt, with examples retrieved based on semantic similarity used in the following template:

```
{ORIGINAL_LLM_PROMPT_FOR_TARGET_TASK}

### Reference Examples (Similar Tasks)
```

```
Here are examples of how to decompose similar tasks. Use the same JSON
    format.

### Example Task: {train_task_1}
---
#### Context
... formatted dynamic_content ...
#### Required Output Plan
... output JSON ...
---

### Example Task: {train_task_2}
---
#### Context
...
#### Required Output Plan
...
---

... (up to K examples) ...

### End of Examples
Now, generate the plan for the **Target Task** defined above.
```

E.2.3 Action Sequencing

No custom prompts used for the final submission.

E.2.4 Transition Modeling

For transition modeling we do SFT and inference on a per action basis.

```
Prompt: <|im_start|>user
{task_name}<|im_end|>
<|im_start|>assistant

Output: <think>

</think>

(:action {task_name}
:parameters ({parameters})
:precondition ({precondition})
:effect ({effect})
)
```

E.3 Other Notable Prompts

Prompts used in prompt engineering methods, written by GPT5-pro and manually refined.

E.3.1 BEHAVIOR GI - VirtualHome-like prompt - 2 shot

```
## Background Introduction
You are a helpful assistant for **goal interpretation** in an embodied
    environment. In the first step: your task is to read a natural-
    language goal (with the task name, goal instructions), the **relevant
     objects** (and their **possible unary states**), and the **initial
    states**, then output the **symbolic version of the goal states**.
```

~~~text
The output consists of two lists:
- **node goals**  unary states for single objects.
- **edge goals**  binary relationships between two objects.

You must output **only JSON** (one object) in the exact format specified
    below.

After this you will be given the PDDL solution, you will then need to
    refine your output based on the PDDL solution to be absolutely
    correct.

But first:

---

## Input Specification
You will receive:
1. **Relevant objects in the scene**  lines of the form:
   `object_name: [list of possible unary states]`
   Example: `candle.n.01_1: ['Dusty']`
   **Allowed unary states** (Behavior):
   `{ "Cooked", "Open", "Frozen", "Dusty", "Stained", "Sliced", "Soaked",
       "Toggled_On", "Burnt" }`
2. **All initial states in the scene**  a list of binary relations (and
    sometimes unary facts) describing the starting world. Example:
   `['ontop', 'cookie.n.01_1', 'table.n.02_1']`
3. **Task Name and Goal Instructions**  free-form natural language
    describing desired outcomes.

> **Note:** Initial states are informative context only; they do **not**
    constrain what can be predicted as a goal. Your job is to interpret
    the **goal instructions**, not to compute a plan.

---

## Output Schema (strict)
Return **exactly** one JSON object string with keys:
```json
{
 "node goals": [ NODE_GOAL, ... ],
 "edge goals": [ EDGE_GOAL, ... ]
}
```
- A **NODE_GOAL** is either `["StateName", "object_name"]` or `["not", ["
    StateName", "object_name"]]`.
 - `StateName`  { "Cooked", "Open", "Frozen", "Dusty", "Stained", "Sliced
      ", "Soaked", "Toggled_On", "Burnt" }.
 - `object_name` must be one of the **relevant objects** provided.
- An **EDGE_GOAL** is either `["Relation", "obj_a", "obj_b"]` or `["not",
     ["Relation", "obj_a", "obj_b"]]`.
 - `Relation`  { "NextTo", "Inside", "OnFloor", "Touching", "Under", "
      OnTop" }.
 - `obj_a` and `obj_b` must be from **relevant objects** provided.

**No other keys or text** may be emitted. Do **not** include explanations.


---
~~~

~~~~
## Quantifier Grounding & Deterministic Pairing
When the instructions include **quantifiers or counting** (e.g., one of
    each into each basket, all X into Y):
1. **Enumerate** the involved objects from the relevant-objects list in
    **lexicographic order**.
2. **Ground** the goal into **fully instantiated** facts (no quantifiers
    in the output).
3. For **balanced assignments** (e.g., one cookie per basket), use **
    stable indexmatching**: the *i*th item of a type maps to the *i*th
    target container, cycling only if counts differ.

---

## Examples

### Example 1  assembling_gift_baskets
**Relevant objects in the scene**
```
basket.n.01_1: ['Stained', 'Dusty']
basket.n.01_2: ['Stained', 'Dusty']
basket.n.01_3: ['Stained', 'Dusty']
basket.n.01_4: ['Stained', 'Dusty']
floor.n.01_1: ['Stained', 'Dusty']
candle.n.01_1: ['Dusty']
candle.n.01_2: ['Dusty']
candle.n.01_3: ['Dusty']
candle.n.01_4: ['Dusty']
cookie.n.01_1: ['Frozen', 'Cooked', 'Burnt']
cookie.n.01_2: ['Frozen', 'Cooked', 'Burnt']
cookie.n.01_3: ['Frozen', 'Cooked', 'Burnt']
cookie.n.01_4: ['Frozen', 'Cooked', 'Burnt']
cheese.n.01_1: ['Frozen', 'Cooked', 'Burnt']
cheese.n.01_2: ['Frozen', 'Cooked', 'Burnt']
cheese.n.01_3: ['Frozen', 'Cooked', 'Burnt']
cheese.n.01_4: ['Frozen', 'Cooked', 'Burnt']
bow.n.08_1: ['Dusty']
bow.n.08_2: ['Dusty']
bow.n.08_3: ['Dusty']
bow.n.08_4: ['Dusty']
table.n.02_1: ['Stained', 'Dusty']
table.n.02_2: ['Stained', 'Dusty']
```

**All initial states in the scene**
```
['onfloor', 'basket.n.01_1', 'floor.n.01_1']
['onfloor', 'basket.n.01_2', 'floor.n.01_1']
['onfloor', 'basket.n.01_3', 'floor.n.01_1']
['onfloor', 'basket.n.01_4', 'floor.n.01_1']
['ontop', 'candle.n.01_1', 'table.n.02_1']
['ontop', 'candle.n.01_2', 'table.n.02_1']
['ontop', 'candle.n.01_3', 'table.n.02_1']
['ontop', 'candle.n.01_4', 'table.n.02_1']
['ontop', 'cookie.n.01_1', 'table.n.02_1']
['ontop', 'cookie.n.01_2', 'table.n.02_1']
['ontop', 'cookie.n.01_3', 'table.n.02_1']
['ontop', 'cookie.n.01_4', 'table.n.02_1']
['ontop', 'cheese.n.01_1', 'table.n.02_2']
~~~~

~~~
[’ontop’, ’cheese.n.01_2’, ’table.n.02_2’]
[’ontop’, ’cheese.n.01_3’, ’table.n.02_2’]
[’ontop’, ’cheese.n.01_4’, ’table.n.02_2’]
[’ontop’, ’bow.n.08_1’, ’table.n.02_2’]
[’ontop’, ’bow.n.08_2’, ’table.n.02_2’]
[’ontop’, ’bow.n.08_3’, ’table.n.02_2’]
[’ontop’, ’bow.n.08_4’, ’table.n.02_2’]
[’onfloor’, ’agent.n.01_1’, ’floor.n.01_1’]
```

**Task Name and Goal Instructions**
```
{
 "Task Name": "assembling_gift_baskets",
 "Goal Instructions": "Put one candle, one cheese, one cookie, and one
     bow into each basket."
}
```

**Output (symbolic goals)**
```json
{
 "node goals": [],
 "edge goals": [
   ["Inside", "candle.n.01_1", "basket.n.01_1"],
   ["Inside", "cheese.n.01_1", "basket.n.01_1"],
   ["Inside", "cookie.n.01_1", "basket.n.01_1"],
   ["Inside", "bow.n.08_1", "basket.n.01_1"],

   ["Inside", "candle.n.01_2", "basket.n.01_2"],
   ["Inside", "cheese.n.01_2", "basket.n.01_2"],
   ["Inside", "cookie.n.01_2", "basket.n.01_2"],
   ["Inside", "bow.n.08_2", "basket.n.01_2"],

   ["Inside", "candle.n.01_3", "basket.n.01_3"],
   ["Inside", "cheese.n.01_3", "basket.n.01_3"],
   ["Inside", "cookie.n.01_3", "basket.n.01_3"],
   ["Inside", "bow.n.08_3", "basket.n.01_3"],

   ["Inside", "candle.n.01_4", "basket.n.01_4"],
   ["Inside", "cheese.n.01_4", "basket.n.01_4"],
   ["Inside", "cookie.n.01_4", "basket.n.01_4"],
   ["Inside", "bow.n.08_4", "basket.n.01_4"]
 ]
}
```

---

### Example 2  dust_free_display
**Relevant objects in the scene**
```
(carry over the same list as Example 1)
```

**All initial states in the scene**
```
(carry over the same list as Example 1)
```
~~~

~~~
**Task Name and Goal Instructions**
```
{
 "Task Name": "dust_free_display",
 "Goal Instructions": "Place all candles on top of the table and ensure
    no candle is dusty. Place all bows on top of table."
}
```
> It is possible some instructions might be in first person ("your"),
  second person ("my") or other forms. Always interpret them as
  referring to the goals for you in the scene.

**Output (symbolic goals)**
```json
{
 "node goals": [
  ["not", ["Dusty", "candle.n.01_1"]],
  ["not", ["Dusty", "candle.n.01_2"]],
  ["not", ["Dusty", "candle.n.01_3"]],
  ["not", ["Dusty", "candle.n.01_4"]]
 ],
 "edge goals": [
  ["OnTop", "candle.n.01_1", "table.n.02_1"],
  ["OnTop", "candle.n.01_2", "table.n.02_1"],
  ["OnTop", "candle.n.01_3", "table.n.02_1"],
  ["OnTop", "candle.n.01_4", "table.n.02_1"],

  ["OnTop", "bow.n.08_1", "table.n.02_2"],
  ["OnTop", "bow.n.08_2", "table.n.02_2"],
  ["OnTop", "bow.n.08_3", "table.n.02_2"],
  ["OnTop", "bow.n.08_4", "table.n.02_2"]
 ]
}
```

---

Now, generate the subgoal plan for the following task.

# Target Task:

**Relevant objects in the scene**
```
{{RELEVANT_OBJECTS}}
```

**All initial states in the scene**
```
{{INITIAL_STATES}}
```

**Task Name and Goal Instructions**
```
{{TASK_INSTRUCTIONS}}
```

---
~~~

```
## Final Instruction
Now, using the **JSON format** defined above, output **just** the
    symbolic version of the goal states (one JSON object). **Do not**
    include any explanation. **Strictly** follow the symbolic goal format
     state/relationship vocabularies, and rules.
```

E.3.2 BEHAVIOR SD - VirtualHome-style prompt + avoid chaining goals

```
## Background Introduction
You are determining complete state transitions of a household task solved
     by a robot. The goal is to list all intermediate states (subgoals)
    and necessary enabling states in temporal order to achieve the target
     goals. The output consists of **Boolean expressions**, which are
    comprised of **state primitives only**. In short, your task is to
    output the **subgoal plan** in the required format.

---

## Data Vocabulary Introduction

### Available States
A **state primitive** is a tuple of a predicate name and its arguments:
‘<PredicateName>(Params)‘ where ‘<PredicateName>‘ is the state name and
    each param ends with an id.

Below is the complete vocabulary of state primitives that you can and
    only can choose from (Behavior variant, **lowercase**).

| State Name | Arguments | Description |
| --- | --- | --- |
| ‘inside‘ | (obj1.id, obj2.id) | obj1 is inside obj2. If ‘inside(A, B)‘
    and B is openable and **not open**, ensure ‘open(B)‘ before changing ‘
    inside(A, B)‘. ‘inside(obj1, agent)‘ is invalid. |
| ‘ontop‘ | (obj1.id, obj2.id) | obj1 is on top of obj2 |
| ‘nextto‘ | (obj1.id, obj2.id) | obj1 is next to obj2 |
| ‘under‘ | (obj1.id, obj2.id) | obj1 is under obj2 |
| ‘onfloor‘ | (obj1.id, floor2.id) | obj1 is on the floor2 |
| ‘touching‘ | (obj1.id, obj2.id) | obj1 is touching or next to obj2 |
| ‘cooked‘ | (obj1.id) | obj1 is cooked |
| ‘burnt‘ | (obj1.id) | obj1 is burnt |
| ‘dusty‘ | (obj1.id) | obj1 is dusty. To achieve ‘not dusty(obj1)‘,
    either (i) ‘inside(obj1, sink)‘ and ‘toggledon(sink)‘, or (ii) hold
    an appropriate cleaning tool. |
| ‘frozen‘ | (obj1.id) | obj1 is frozen |
| ‘open‘ | (obj1.id) | obj1 is open |
| ‘sliced‘ | (obj1.id) | obj1 is sliced |
| ‘soaked‘ | (obj1.id) | obj1 is soaked |
| ‘stained‘ | (obj1.id) | obj1 is stained. To achieve ‘not stained(obj1)‘,
     either (i) ‘inside(obj1, sink)‘ and ‘toggledon(sink)‘, (ii) ‘soaked(
    cleaner)‘, or (iii) hold detergent. |
| ‘toggledon‘ | (obj1.id) | obj1 is toggled on |
| ‘holds_rh‘ | (obj1.id) | obj1 is in the right hand of the robot |
| ‘holds_lh‘ | (obj1.id) | obj1 is in the left hand of the robot |

### Available Connectives
Use the following logical connectives to build complex expressions **
    within the same time step** if needed.:
```

~~~~
| Connective | Form | Meaning |
| --- | --- | --- |
| `and` | `exp1 and exp2` | true if both are true |
| `or` | `exp1 or exp2` | true if either is true |
| `not` | `not exp` | true if `exp` is false |

> Advanced quantifiers may appear **in goal specifications** in the
    dataset (e.g., `forall`, `exists`, `forn`, `fornpairs`), but **you
    must not output quantifiers**. Always **instantiate** quantified
    goals over the concrete objects listed in the scene and produce only
    grounded atoms in the final plan.

---

## Rules You Must Follow
- The initial states are given at the beginning of the task.
- **Temporal order:** Each list item in `output` is a later step than the
     previous one. Boolean expressions **on the same line** are
    interchangeable and simultaneous (use `and`).
- Avoid using `or`/ `and` unecessarily within a single step unless
    absolutely needed. Even if multiple states can be achieved
    simultaneously, it's often clearer to list them in separate steps.
- Add intermediate states if necessary to improve logical consistency.
- If you want to change the state of `A` while `A` is inside `B` **and** `
    B` is openable and not open, ensure `open(B)` first.
- The robot can hold one object per hand. To relocate an object, include
    an appropriate `holds_rh/holds_lh` state first.
- Do not output `inside(obj, agent)` (invalid).
- Do not output redundant states (already true and never invalidated).
- Think about the effects of certain actions on the objects' states. (e.g
    ., slicing an object will result in it breaking into pieces, after
    which it cannot be picked up as a whole.) Figure out the correct
    sequence of actions accordingly
- **Quantified goals:** If goals use `forall/exists/forn/fornpairs`, **
    instantiate** them to concrete objects; **do not** output quantifiers.


---

## Output Format (JSON)
Your output must strictly follow:
```json
{"output": [<your subgoal plan>]}
```
- `output` is a list of Boolean expressions (strings) in **temporal order
    **.
- Use only the **states** from the table above and the connectives `and`,
     `or`, `not` **within a single step**.

---

Now, generate the subgoal plan for the following task.

# Target Task: {{TASK_NAME}}

## Relevant objects in this scene
```
{{RELEVANT_OBJECTS}}
```
~~~~

```
## Initial States
```
```
{{INITIAL_STATES}}
```
```
## Goal States
```
```
{{GOAL_STATES}}
```
```
---

## Final Instruction
Based on the initial states in a task, achieve the final goal states
    logically and reasonably. Make sure your output follows the JSON
    format above exactly. **Do not reply with anything else**; use only
    the listed **states**, with `and`, `or`, and `not` within a single
    time step if required.
```

E.3.3 BEHAVIOR AS - Strong location assertion

```
Problem:
You are designing instructions for a household robot.
The goal is to guide the robot to modify its environment from an initial
    state to a desired final state.
The input will be the initial environment state, the target environment
    state, the objects you can interact with in the environment.
The output should be a list of action commands so that after the robot
    executes the action commands sequentially, the environment will
    change from the initial state to the target state.

Data format: After # is the explanation.

Format of the states:
The environment state is a list starts with a uniary predicate or a
    binary prediate, followed by one or two obejcts.
You will be provided with multiple environment states as the initial
    state and the target state.
For example:
['inside', 'strawberry_0', 'fridge_97'] #strawberry_0 is inside fridge_97
['not', 'sliced', 'peach_0'] #peach_0 is not sliced
['ontop', 'jar_1', 'countertop_84'] #jar_1 is on top of countertop_84

Format of the action commands:
Action commands is a dictionary with the following format:
{
      "action": "action_name",
      "object": "target_obj_name",
}

or

{
      "action": "action_name",
      "object": "target_obj_name1,target_obj_name2",
}

The action_name must be one of the following:
```

```
LEFT_GRASP # the robot grasps the object with its left hand, to execute
    the action, the robot’s left hand must be empty, e.g. {’action’: ’
    LEFT_GRASP’, ’object’: ’apple_0’}.
RIGHT_GRASP # the robot grasps the object with its right hand, to execute
     the action, the robot’s right hand must be empty, e.g. {’action’: ’
    RIGHT_GRASP’, ’object’: ’apple_0’}.
LEFT_PLACE_ONTOP # the robot places the object in its left hand on top of
     the target object and release the object in its left hand, e.g. {’
    action’: ’LEFT_PLACE_ONTOP’, ’object’: ’table_1’}.
RIGHT_PLACE_ONTOP # the robot places the object in its right hand on top
    of the target object and release the object in its left hand, e.g. {’
    action’: ’RIGHT_PLACE_ONTOP’, ’object’: ’table_1’}.
LEFT_PLACE_INSIDE # the robot places the object in its left hand inside
    the target object and release the object in its left hand, to execute
     the action, the robot’s left hand must hold an object, and the
    target object can’t be closed e.g. {’action’: ’LEFT_PLACE_INSIDE’, ’
    object’: ’fridge_1’}.
RIGHT_PLACE_INSIDE # the robot places the object in its right hand inside
     the target object and release the object in its left hand, to
    execute the action, the robot’s right hand must hold an object, and
    the target object can’t be closed, e.g. {’action’: ’
    RIGHT_PLACE_INSIDE’, ’object’: ’fridge_1’}.
RIGHT_RELEASE # the robot directly releases the object in its right hand,
     to execute the action, the robot’s left hand must hold an object, e.
    g. {’action’: ’RIGHT_RELEASE’, ’object’: ’apple_0’}.
LEFT_RELEASE # the robot directly releases the object in its left hand,
    to execute the action, the robot’s right hand must hold an object, e.
    g. {’action’: ’LEFT_RELEASE’, ’object’: ’apple_0’}.
OPEN # the robot opens the target object, to execute the action, the
    target object should be openable and closed, also, toggle off the
    target object first if want to open it, e.g. {’action’: ’OPEN’, ’
    object’: ’fridge_1’}.
CLOSE # the robot closes the target object, to execute the action, the
    target object should be openable and open, e.g. {’action’: ’CLOSE’, ’
    object’: ’fridge_1’}.
COOK # the robot cooks the target object, to execute the action, the
    target object should be put in a pan, e.g. {’action’: ’COOK’, ’object
    ’: ’apple_0’}.
CLEAN # the robot cleans the target object, to execute the action, the
    robot should have a cleaning tool such as rag, the cleaning tool
    should be soaked if possible, or the target object should be put into
     a toggled on cleaner like a sink or a dishwasher, e.g. {’action’: ’
    CLEAN’, ’object’: ’window_0’}.
FREEZE # the robot freezes the target object e.g. {’action’: ’FREEZE’, ’
    object’: ’apple_0’}.
UNFREEZE # the robot unfreezes the target object, e.g. {’action’: ’
    UNFREEZE’, ’object’: ’apple_0’}.
SLICE # the robot slices the target object, to execute the action, the
    robot should have a knife in hand, e.g. {’action’: ’SLICE’, ’object’:
     ’apple_0’}.
SOAK # the robot soaks the target object, to execute the action, the
    target object must be put in a toggled on sink, e.g. {’action’: ’SOAK
    ’, ’object’: ’rag_0’}.
DRY # the robot dries the target object, e.g. {’action’: ’DRY’, ’object’:
     ’rag_0’}.
TOGGLE_ON # the robot toggles on the target object, to execute the action,
     the target object must be closed if the target object is openable
    and open e.g. {’action’: ’TOGGLE_ON’, ’object’: ’light_0’}.
```

```
TOGGLE_OFF # the robot toggles off the target object, e.g. {'action': '
    TOGGLE_OFF', 'object': 'light_0'}.
LEFT_PLACE_NEXTTO # the robot places the object in its left hand next to
    the target object and release the object in its left hand, e.g. {'
    action': 'LEFT_PLACE_NEXTTO', 'object': 'table_1'}.
RIGHT_PLACE_NEXTTO # the robot places the object in its right hand next
    to the target object and release the object in its right hand, e.g. {'
    action': 'RIGHT_PLACE_NEXTTO', 'object': 'table_1'}.
LEFT_TRANSFER_CONTENTS_INSIDE # the robot transfers the contents in the
    object in its left hand inside the target object, e.g. {'action': '
    LEFT_TRANSFER_CONTENTS_INSIDE', 'object': 'bow_1'}.
RIGHT_TRANSFER_CONTENTS_INSIDE # the robot transfers the contents in the
    object in its right hand inside the target object, e.g. {'action': '
    RIGHT_TRANSFER_CONTENTS_INSIDE', 'object': 'bow_1'}.
LEFT_TRANSFER_CONTENTS_ONTOP # the robot transfers the contents in the
    object in its left hand on top of the target object, e.g. {'action': '
    LEFT_TRANSFER_CONTENTS_ONTOP', 'object': 'table_1'}.
RIGHT_TRANSFER_CONTENTS_ONTOP # the robot transfers the contents in the
    object in its right hand on top of the target object, e.g. {'action':
     'RIGHT_TRANSFER_CONTENTS_ONTOP', 'object': 'table_1'}.
LEFT_PLACE_NEXTTO_ONTOP # the robot places the object in its left hand
    next to target object 1 and on top of the target object 2 and release
     the object in its left hand, e.g. {'action': '
    LEFT_PLACE_NEXTTO_ONTOP', 'object': 'window_0, table_1'}.
RIGHT_PLACE_NEXTTO_ONTOP # the robot places the object in its right hand
    next to object 1 and on top of the target object 2 and release the
    object in its right hand, e.g. {'action': 'RIGHT_PLACE_NEXTTO_ONTOP',
     'object': 'window_0, table_1'}.
LEFT_PLACE_UNDER # the robot places the object in its left hand under the
     target object and release the object in its left hand, e.g. {'action
    ': 'LEFT_PLACE_UNDER', 'object': 'table_1'}.
RIGHT_PLACE_UNDER # the robot places the object in its right hand under
    the target object and release the object in its right hand, e.g. {'
    action': 'RIGHT_PLACE_UNDER', 'object': 'table_1'}.

Format of the interactable objects:
Interactable object will contain multiple lines, each line is a
    dictionary with the following format:
{
   "name": "object_name",
   "category": "object_category"
}
object_name is the name of the object, which you must use in the action
    command, object_category is the category of the object, which
    provides a hint for you in interpreting initial and goal condtions.

Please pay specail attention:
1. The robot can only hold one object in each hand.
2. Action name must be one of the above action names, and the object name
     must be one of the object names listed in the interactable objects.
3. All PLACE actions will release the object in the robot's hand, you don'
    t need to explicitly RELEASE the object after the PLACE action.
4. For LEFT_PLACE_NEXTTO_ONTOP and RIGHT_PLACE_NEXTTO_ONTOP, the action
    command are in the format of {'action': 'action_name', 'object': '
    obj_name1, obj_name2'}
5. If you want to perform an action to an target object, you must make
    sure the target object is not inside a closed object.
6. For actions like OPEN, CLOSE, SLICE, COOK, CLEAN, SOAK, DRY, FREEZE,
    UNFREEZE, TOGGLE_ON, TOGGLE_OFF, at least one of the robot's hands
    must be empty, and the target object must have the corresponding
    property like they're openable, toggleable, etc.
7. For PLACE actions and RELEASE actions, the robot must hold an object
    in the corresponding hand.
8. Before slicing an object, the robot can only interact with the object
    (e.g. peach_0), after slicing the object, the robot can only interact
     with the sliced object (e.g. peach_0_part_0).
9. Use actions to reassert location predicates at the end of the task,
    even if their initial states were already at the target states.


Examples: after# is the explanation.

Example 1:
Input:
initial environment state:
['stained', 'sink_7']
['stained', 'bathtub_4']
['not', 'soaked', 'rag_0']
['onfloor', 'rag_0', 'room_floor_bathroom_0']
['inside', 'rag_0', 'cabinet_1']
['not', 'open', 'cabinet_1']


target environment state:
['not', 'stained', 'bathtub_4']
['not', 'stained', 'sink_7']
['and', 'soaked', 'rag_0', 'inside', 'rag_0', 'bucket_0']


interactable objects:
{'name': 'sink_7', 'category': 'sink.n.01'}
{'name': 'bathtub_4', 'category': 'bathtub.n.01'}
{'name': 'bucket_0', 'category': 'bucket.n.01'}
{'name': 'rag_0', 'category': 'rag.n.01'}
{'name': 'cabinet_1', 'category': 'cabinet.n.01'}


Please output the list of action commands (in the given format) so that
    after the robot executes the action commands sequentially, the
    current environment state will change to target environment state.
    Usually, the robot needs to execute multiple action commands
    consecutively to achieve final state. Please output multiple action
    commands rather than just one. Only output the list of action
    commands with nothing else.

Output:
[
   {
      "action": "OPEN",
      "object": "cabinet_1"
   }, # you want to get the rag_0 from cabinet_1, should open it first
   {
      "action": "RIGHT_GRASP",
      "object": "rag_0"
   }, # you want to clean the sink_7 and bathtub_4, you found them
       stained, so you need to soak the rag_0 first
   {
      "action": "RIGHT_PLACE_INSIDE",
      "object": "sink_7"
   }, # to soak the rag_0, you need to place it inside the sink_7
   {
      "action": "TOGGLE_ON",
      "object": "sink_7"
   }, # to soak the rag_0, you need to toggle on the sink_7
   {
      "action": "SOAK",
      "object": "rag_0"
   }, # now you can soak the rag_0
   {
      "action": "TOGGLE_OFF",
      "object": "sink_7"
   }, # after soaking the rag_0, you need to toggle off the sink_7
   {
      "action": "LEFT_GRASP",
      "object": "rag_0"
   }, # now you can grasp soaked rag_0 to clean stain
   {
      "action": "CLEAN",
      "object": "sink_7"
   }, # now you clean the sink_7
   {
      "action": "CLEAN",
      "object": "bathtub_4"
   }, # now you clean the bathtub_4
   {
      "action": "LEFT_PLACE_INSIDE",
      "object": "bucket_0"
   } # after cleaning the sink_7, you need to place the rag_0 inside the
       bucket_0
]

Your task:
Input:
initial environment state:
{{INITIAL_STATES}}

target environment state:
{{TARGET_STATES}}

interactable objects:
{{INTERACTABLE_OBJECTS}}

Please output the list of action commands (in the given format) so that
    after the robot executes the action commands sequentially, the
    current environment state will change to target environment state.
    Usually, the robot needs to execute multiple action commands
    consecutively to achieve final state. Please output multiple action
    commands rather than just one. Only output the list of action
    commands with nothing else.

Output:
```

E.3.4 BEHAVIOR TM - Prompt rewrite with GPT-5

```
You are given the domains types and predicates below. **Use them exactly
    as written** (names, arities, and types). Do not invent new
    predicates, types, or objects.

(define (domain igibson)
   (:requirements :strips :adl :typing :negative-preconditions)
   (:types
      vacuum_n_04 facsimile_n_02 dishtowel_n_01 apparel_n_01 seat_n_03
          bottle_n_01 mouse_n_04 window_n_01 scanner_n_02
      sauce_n_01 spoon_n_01 date_n_08 egg_n_02 cabinet_n_01 yogurt_n_01
          parsley_n_02 notebook_n_01 dryer_n_01 saucepan_n_01
      soap_n_01 package_n_02 headset_n_01 fish_n_02 vehicle_n_01
          chestnut_n_03 grape_n_01 wrapping_n_01 makeup_n_01 mug_n_04
      pasta_n_02 beef_n_02 scrub_brush_n_01 cracker_n_01 flour_n_01
          sunglass_n_01 cookie_n_01 bed_n_01 lamp_n_02 food_n_02
      painting_n_01 carving_knife_n_01 pop_n_02 tea_bag_n_01 sheet_n_03
          tomato_n_01 agent_n_01 hat_n_01 dish_n_01 cheese_n_01
      perfume_n_02 toilet_n_02 broccoli_n_02 book_n_02 towel_n_01
          table_n_02 pencil_n_01 rag_n_01 peach_n_03 water_n_06 cup_n_01
      radish_n_01 marker_n_03 tile_n_01 box_n_01 screwdriver_n_01
          raspberry_n_02 banana_n_02 grill_n_02 caldron_n_01
          vegetable_oil_n_01
      necklace_n_01 brush_n_02 washer_n_03 hamburger_n_01 catsup_n_01
          sandwich_n_01 plaything_n_01 candy_n_01 cereal_n_03 door_n_01
      food_n_01 newspaper_n_03 hanger_n_02 carrot_n_03 salad_n_01
          toothpaste_n_01 blender_n_01 sofa_n_01 plywood_n_01 olive_n_04
          briefcase_n_01
      christmas_tree_n_05 bowl_n_01 casserole_n_02 apple_n_01 basket_n_01
           pot_plant_n_01 backpack_n_01 sushi_n_01 saw_n_02
          toothbrush_n_01
      lemon_n_01 pad_n_01 receptacle_n_01 sink_n_01 countertop_n_01
          melon_n_01 bracelet_n_02 modem_n_01 pan_n_01 oatmeal_n_01
          calculator_n_02
      duffel_bag_n_01 sandal_n_01 floor_n_01 snack_food_n_01
          stocking_n_01 dishwasher_n_01 pencil_box_n_01 chicken_n_01
          jar_n_01 alarm_n_02
      stove_n_01 plate_n_04 highlighter_n_02 umbrella_n_01
          piece_of_cloth_n_01 bin_n_01 ribbon_n_01 chip_n_04 shelf_n_01
          bucket_n_01 shampoo_n_01
      folder_n_02 shoe_n_01 detergent_n_02 milk_n_01 beer_n_01 shirt_n_01
           dustpan_n_02 cube_n_05 broom_n_01 candle_n_01 pen_n_01
          microwave_n_02
      knife_n_01 wreath_n_01 car_n_01 soup_n_01 sweater_n_01 tray_n_01
          juice_n_01 underwear_n_01 orange_n_01 envelope_n_01 fork_n_01
          lettuce_n_03
      bathtub_n_01 earphone_n_01 pool_n_01 printer_n_03 sack_n_01
          highchair_n_01 cleansing_agent_n_01 kettle_n_01
          vidalia_onion_n_01 mousetrap_n_01
      bread_n_01 meat_n_01 mushroom_n_05 cake_n_03 vessel_n_03 bow_n_08
          gym_shoe_n_01 hammer_n_02 teapot_n_01 chair_n_01 jewelry_n_01
          pumpkin_n_02 sugar_n_01
      shower_n_01 ashcan_n_01 hand_towel_n_01 pork_n_01 strawberry_n_01
          electric_refrigerator_n_01 oven_n_01 ball_n_01 document_n_01
          sock_n_01 beverage_n_01
      hardback_n_01 scraper_n_01 carton_n_02
      agent
   )
   (:predicates
      (inside ?obj1 - object ?obj2 - object)
      (nextto ?obj1 - object ?obj2 - object)
      (ontop ?obj1 - object ?obj2 - object)
      (under ?obj1 - object ?obj2 - object)
      (cooked ?obj1 - object)
      (dusty ?obj1 - object)
      (frozen ?obj1 - object)
      (open ?obj1 - object)
      (stained ?obj1 - object)
      (sliced ?obj1 - object)
      (soaked ?obj1 - object)
      (toggled_on ?obj1 - object)
      (onfloor ?obj1 - object ?floor1 - object)
      (holding ?obj1 - object)
      (handsfull ?agent1 - agent)
      (in_reach_of_agent ?obj1 - object)
      (same_obj ?obj1 - object ?obj2 - object)
   )
   ;; Actions to be predicted
)

# Objective
Given a **problem file** (objects, :init, :goal) and a list of **
    unfinished actions** (names + parameters), **write only the :
    precondition and :effect** for each action in valid PDDL.

# Hard rules (optimize for planner success, then F1)
1) **Use only the given predicates and types** exactly as declared above.
     Do **not** invent new predicates or rename existing ones.
2) **Scope safety:** Every variable that appears in a precondition or
    effect **must** be listed in that actions `:parameters`. Do **not**
    introduce new variables.
3) **Local, not global:** Do **not** use `forall` or effects that change
    unrelated objects. Effects must only mention the actions parameters.
4) **Preconditions = DNF only** using `and`, `or`, `not`. Keep them
    satisfiable (avoid over-constraining).
5) **Effects = minimal AND of changes.** Prefer unconditional effects.
    Use `when` only if necessary and only over the actions parameters. No
     `or` in effects.
6) **Flip at least one key predicate per action** (e.g., add `(
    in_reach_of_agent ?o)` in `navigate_to`, add `(holding ?o)` in `grasp
    `, add `(open ?c)` in `open`, add `(inside ?o ?c)` or `(ontop ?o ?s)`
     in `place_*` and delete `(holding ?o)` if the action puts the object
     down).
7) **No global resets / side effects** (e.g., do not clear reachability
    for all other objects).
8) **Typing must match** the parameters provided; keep syntax strictly
    valid for classical planners.

# Output format (strict)
- For each action, output a single PDDL block:
(:action <name>
:parameters (<params>)
:precondition (<DNF>)
:effect (<AND-of-effects>)
)

- Concatenate all action blocks **in order** into one string.
- Return **JSON** with a single key `"output"` whose value is that
    concatenated string: `{"output": "..."}`.
- Do not add commentary.
```

```
# Small example (illustrative only; uses only declared predicates)
Input:
Problem file:
(define (problem cleaning_floor_0)
(:domain igibson)
(:objects floor_n_01_1 - floor_n_01
       rag_n_01_1 - rag_n_01
       sink_n_01_1 - sink_n_01
       agent_1 - agent)
(:init (dusty floor_n_01_1) (stained floor_n_01_1) (toggled_on
    sink_n_01_1))
(:goal (and (not (dusty floor_n_01_1)) (not (stained floor_n_01_1))))
)
Actions to be finished:
(:action navigate_to
:parameters (?obj - object ?agent - agent)
:precondition ()
:effect ())
(:action soak-rag
:parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
:precondition ()
:effect ())
(:action clean-stained-floor-rag
:parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
:precondition ()
:effect ())
(:action clean-dusty-floor-rag
:parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
:precondition ()
:effect ())

Output:
{"output":"(:action navigate_to
:parameters (?obj - object ?agent - agent)
:precondition (not (in_reach_of_agent ?obj))
:effect (in_reach_of_agent ?obj)
)
(:action soak-rag
:parameters (?rag - rag_n_01 ?sink - sink_n_01 ?agent - agent)
:precondition (and (holding ?rag) (in_reach_of_agent ?sink) (toggled_on ?
    sink))
:effect (soaked ?rag)
)
(:action clean-stained-floor-rag
:parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
:precondition (and (stained ?floor) (holding ?rag) (in_reach_of_agent ?
    floor))
:effect (not (stained ?floor))
)
(:action clean-dusty-floor-rag
:parameters (?rag - rag_n_01 ?floor - floor_n_01 ?agent - agent)
:precondition (and (dusty ?floor) (holding ?rag) (in_reach_of_agent ?
    floor))
:effect (not (dusty ?floor))
)"}

# Now do the real task
Input:
Problem file:
```

```
{{PROBLEM_FILE}}

Action to be finished:
{{ACTIONS}}

Output:
```

E.3.5 VIRTUALHOME GI - 2 step prompt

```
# Prompt Template: virtualhome_goal_interpretation

# SYSTEM PURPOSE
Produce a natural-language interpretation and plan for a household-robot
    goal.
Focus on **visualizing the final state of the environment**describe what
    the world looks like when the goal is fully achieved.
This interpretation anticipates later conversion into **node goals**, **
    edge goals**, and **action goals**, but does not output JSON.

# OUTPUT SHAPE
Write the sections below in this exact order; no JSON, questions, or meta-
    commentary.

# INPUT
<CONTEXT>
RELEVANT_OBJECTS:
{{RELEVANT_OBJECTS}}

GOAL:
{{GOAL_SECTION}}
</CONTEXT>

# MENTAL MODEL (for reasoning; do not copy into output)
- Imagine the scene *after* success: what objects exist, how are they
    positioned, what are their final states or relationships. Walk
    thtrough the steps needed to reach that state.
- **Node goals:** final states of objects (only from allowed enums).
    Repetition of an initial state is fine if that state must hold true
    in the end to achieve the main goal.
- **Edge goals:** stable spatial or ownership relations in the final
    world snapshot.
- **Action goals:** actions that are *ends in themselves* (e.g., TURN ON
    lamp), not just transitions needed to reach a state (e.g., OPEN
    fridge to get food).
- Distinguish ON(state) vs ON(relation). ON can be used for edge goals.
- Certain relations goals can be used even if they don't gramatically fit
     naturally (e.g., SOAP ON WashingMachine to mean SOAP is in the
    WashingMachine). If you can't find a better way to express a needed
    relation, use this (up to an extent).

# CONSTRAINTS
- Only use object names appearing in RELEVANT_OBJECTS.
- Allowed Node State enum: CLOSED, OPEN, ON, OFF, SITTING, DIRTY, CLEAN,
    LYING, PLUGGED_IN, PLUGGED_OUT.
- Maximum of 2 action goals.
- Prefer **final-state completeness** over minimalisminclude all states
    that must hold simultaneously.
- Length: 615 bullets (200 words).
```

```
# REQUIRED SECTIONS
1) **Goal summary (12 sentences).**
2) **Plan (ordered steps)**  what happens to reach that final visualized
    state.
3) **Node goals (natural language)**  list of desired end-states with
    enum hints (UPPERCASE).
4) **Edge goals (natural language)**  list using FROM RELATION TO.
5) **Action goals (natural language)**  only those actions that *are*
    explicit goals.

# STYLE
- Be concise, factual, and scene-focused.
- Use enums in parentheses.

# EXAMPLE OUTPUT SHAPE
Goal summary:
Plan:
-
Node goals:
-
Edge goals:
-
Action goals:
-
```

Followed by:

```
Great! Now we will move to the next step.
Given the context below, correct any mistakes if you find any from your
    previous response. Then, produce **only** the required symbolic goals
     JSON.
Visualize the **final environment state** that represents the goal being
    achieved, and convert that visualization into symbolic goals.


# ADDITIONAL CONSTRAINTS

## Relation  Allowed 'to_name' Targets

| Relation | Allowed Targets |
|-----------|----------------|
| **ON** | table, character, dishwasher, toilet, oven, couch, bed,
    washing_machine, coffe_maker |
| **HOLDS_LH** | water_glass, novel, tooth_paste, keyboard, spectacles,
    toothbrush |
| **HOLDS_RH** | phone, mouse, water_glass, remote_control, address_book,
     novel, tooth_paste, cup, drinking_glass, toothbrush |
| **INSIDE** | home_office, hands_both, freezer, bathroom, dining_room |
| **FACING** | phone, toilet, television, computer, laptop,
    remote_control |
| **CLOSE** | shower, cat |


# GUIDELINES

!!Node goals are the final states when the task is done. If the goal is
    turn on the lamp, the node goal is that the lamp is ON, and there is
    no action goal needed. But if the goal is write an email, then an
    action goal TYPE is needed because the action is the goal itself; you
     should also need shut down the computer after writing the email.
    Visualize the completed scenesimulate what the robot would see at the
     endto decide which states must hold!!

# INSTRUCTIONS
Using only the CONTEXT and GOAL above, output the symbolic version of the
     goals as a JSON object with the three required keys.
- Derive **node goals** only with the allowed node states.
- Derive **edge goals** only when the relation and to_name are permitted
    by the relationtarget constraints.
- Use **action goals** (2) to capture essential steps not expressible as
    valid node/edge goals under these constraints.

# OUTPUT FORMAT (strict)
Return exactly one JSON object with these toplevel keys in this order:
{"node goals": [...], "edge goals": [...], "action goals": [...]}
No extra text, explanations, or code fences.

**Symbolic goals JSON  formats**

Top-level shape:
{
 "node goals": [NodeGoal, ...],
 "edge goals": [EdgeGoal, ...],
 "action goals": [ActionGoal, ...]
}

NodeGoal:
{ "name": "<object_name>", "state": "<STATE_ENUM>" }
STATE_ENUM  {CLOSED, OPEN, ON, OFF, SITTING, DIRTY, CLEAN, LYING,
    PLUGGED_IN, PLUGGED_OUT}

EdgeGoal:
{ "from_name": "<object_name>", "relation": "<REL_ENUM>", "to_name": "<
    object_name>" }
REL_ENUM  {ON, INSIDE, BETWEEN, CLOSE, FACING, HOLDS_RH, HOLDS_LH}
Constraint: to_name must be an allowed target for relation (per provided
    relationtargets mapping).

ActionGoal:
{ "action": "<ACTION_ENUM>", "description": "<short phrase>" }
Limit: 02 actions total (use only if node/edge goals cant fully capture
    the intent).

ONLY output the JSON object as specified above. Do not include any other
    text.
```