A Dataset Construction

A.1 Building the Needle(s) in the Embodied Haystack Benchmark

We construct the Needle(s) in the Embodied Haystack benchmark in three stages: 1) Trajectory Replay and Metadata Collection; 2) Rule-Based QA Generation; and 3) Cross-validation with Multimodal LLMs. The following sections provide detailed descriptions of each step.

A.1.1 Trajectory Replay and Metadata Collection

We first replay 225 test trajectories generated by ∞-THOR, logging both visual observations (agent's egocentric views) and structured metadata at each timestep. For every step, we store the list of visible objects, agent-inventory items, openable containers, and their contents from the simulator. This produces a fine-grained interaction log that captures grounded scene dynamics over time.

An example of the collected metadata at a single timestep is shown below:

Example of metadata entry


```
{
    "img_idx": 2,
    "img_filename": "000000002.png",
    "step": 1,
    "object_log": {
        "visible": ["Shelf", "Vase", "Book"],
        "pickupable": ["Vase", "Book"],
        "isOpen": [],
        "inven_obj": [],
        "receptacles": ["Shelf"],
        "recep_objs": {
            "Shelf": ["Vase", "Book"]
        }
    }
}
```


Each metadata entry corresponds to a low-level action step and provides the semantic state of the scene, enabling the construction of temporally grounded QA instances in later stages.

A.1.2 Rule-Based QA Generation

To construct the QA set, we apply rule-based generation templates to each trajectory using its sequence of low-level actions and associated metadata. The QA generation process involves parsing the agent's interactions with objects, containers, and the environment, and applying a set of handcrafted rules to synthesize grounded questions.

Our QA generation logic covers a diverse range of question types, including object presence, object state, location tracking, slicing actions, container content reasoning, and action counting. For instance, if an object is seen for the first time at a particular step, a presence question such as "Is there any `apple` in this room?" is generated. Similarly, after a `PutObject` action, location-based questions like "Where was the `apple` before you put it to the `microwave`?" are produced. When slicing actions happen, we create questions about the object being sliced and other nearby items (e.g., "What objects were in the `Fridge` when you sliced the `apple`?"). Then, we sample questions based on the frequency to ensure diversity across object types, and annotate the GT answer steps using the replay logs. Table 3 summarizes the types of questions generated, and corresponding trigger conditions and example templates.

A.1.3 Cross-validation with Multimodal LLMs

To ensure the answerability and clarity of the generated QA pairs, we perform cross-validation using four powerful multimodal LLMs: LLaVA-OneVision 7B (Li et al., 2024a), Qwen2.5-VL 7B (Bai et al., 2025), Deepseek-VL 7B (Lu et al., 2024), and Pixtral 12B (Agrawal et al., 2024). Each model

Table 3: QA types, trigger conditions, and corresponding question templates used in rule-based generation.

QA Type	Trigger Condition	Example Template(s)
object presence (Yes/No)	object appears visibly in the trajectory	`Is there any {obj} in this room?` `Have you seen a/an {obj}?`
open state questions	container marked as open in metadata	`Was {container} open?`

object location tracing	sequences of `Pickup` and `PutObject` actions	`Where was {obj} before you put it to {container}?`
		`Where did you move the {obj} from the {container}?`
		`Where is {obj} now?`

slicing-based questions	`SliceObject` action detected in trajectory	`What did you slice?` `What objects were in/on the {container} when you slice the {obj}?`
container content	container visibility with non-empty contents	`What objects were in/on the {container}?` `What object did you put in/on the {container}?`
put action questions	unique `PutObject` action for a container	`What object did you put in/on the {container}?`

final object state	final location of an object at episode end	`Is {obj} in/on the {container}?`
		`What objects are in/on the {container}?`
		`How many objects were in/on the {container}?`

movement counting	object picked up more than once	`How many times did you move {obj}?`

is prompted with the GT images corresponding to the annotated QA steps and asked to answer the associated questions. Given their strong performance on standard visual QA tasks, we use these models to assess whether a question can be correctly answered or not. We keep only the QA pairs that are correctly answered by at least one of the four models, and discard those that fail across all models. This helps improve dataset quality and filtering out ambiguous or visually ungroundable questions. Table 4 shows the accuracy of each model on the finalized QA set when evaluated with GT images. Notably, even with access to GT images, all models struggle with questions requiring reasoning over three or more evidence steps. To maintain the benchmark's difficulty and support evaluation of more capable models in future, we manually inspect the multi-clue questions and include those that are answerable.

Table 4: QA accuracy (%) of multimodal LLMs on ground-truth images.

Model	Size	# of clues (GT steps)			Total
		1	2	≥3	
LLaVA-OneVision	7B	86.61	68.55	23.74	71.15
Qwen2.5-VL	7B	85.94	89.83	64.40	82.20
Deepseek-VL	7B	81.56	39.14	22.57	62.88
Pixtral	12B	91.34	39.60	58.56	76.25

A.2 Constructing Long-Horizon Trajectories

To synthesize long-horizon trajectories, we construct each trajectory by sequentially chaining successful sub-tasks sampled from a predefined set of task templates. This process is illustrated in Algorithm 1. We begin by sampling a task template from a fixed task pool, which includes goal types such as `pick and place simple`, `pick two obj and place`, and `pick and place with movable recep`. Each sampled template requires relevant objects in the scene (e.g., pickupable items, target receptacles), which are then used to define the goal for that task.

We use a classical task planner, which operates over PDDL-defined domains (Shridhar et al., 2020), to generate a low-level action sequence for the sampled goal, and simulate this plan in an interactive environment. If the rollout fails (e.g., due to collisions, object occlusions, or unreachable conditions), we discard the sequence and re-sample from the task pool. Otherwise, the successful rollout is retained and appended to the ongoing trajectory.

This sampling-execution loop is repeated until a long trajectory with a desired number of sub-goals is formed. The resulting synthetic long-horizon trajectory consists of multiple sub-goals concatenated into a continuous sequence. To induce long-term temporal dependencies, the final sub-task is constrained to involve only objects that appear in the early 20% and late 20% of the overall trajectory, requiring the agent to integrate temporally distant evidence to answer associated questions.

Algorithm 1 Construct Long-horizon Trajectory

Input: Task Pool $\mathcal{T}$, max sub goals N
Output: Long-horizon trajectory τ
Initialize empty trajectory $\tau \leftarrow []$
while len(τ) $< N$ **do**
    Sample task template $g \sim \mathcal{T}$ and objects
    Plan action sequence π_g by planner
    if Simulate(π_g) is successful **then**
        Append π_g to trajectory: $\tau \leftarrow \tau \parallel \pi_g$
    else
        Discard and re-sample
    end if
end while
// Final sub-task with long-term object dependency
Sample $g_{\text{final}} \sim \mathcal{T}$ and objects in early 20% and late 20%
Plan and simulate π_{final} using restricted objects
if Simulate(π_{final}) is successful **then**
    Append π_{final} to trajectory: $\tau \leftarrow \tau \parallel \pi_{\text{final}}$
else
    Repeat sampling until success
end if
return τ

B Training and Evaluation Details

B.1 Interactive Evaluation in ∞-Thor

Training. We fine-tune the LLaVA-OneVision 7B model on our training set while freezing the vision encoder. The model is trained using a next-action prediction objective, where only the action tokens are optimized, conditioned on the goal and state tokens. Table 5 summarizes the training specifications for different context lengths. For 32K training, we apply tensor parallelism with a degree of 4 and pipeline parallelism with a degree of 2, utilizing 8 H100 GPUs in total. Since pipeline parallelism requires the batch size to match the pipeline degree, we set the batch size to 2. For longer context lengths, we use context parallelism: 8-way for 64K (on 8 GPUs) and 16-way for 130K (on 16 GPUs). All models are fine-tuned for approximately 3 epochs with a learning rate of 1e-5, using the AdamW optimizer and a linear learning rate schedule with a 0.03 warmup ratio.

Table 5: Training specifications for different context lengths.

Context Length	Parallelism	# GPUs	Training Time
32K	Tensor (4) + Pipeline (2)	8	160 hrs
64K	Context (8)	8	120 hrs
130K	Context (16)	16	134 hrs

Plan-Level Evaluation. We evaluate agent performance using a plan-level framework, where each plan corresponds to a short sequence of actions aimed at achieving a specific intermediate sub-goal (e.g., navigating to an object, placing an item). A trajectory is composed of multiple such plans, executed sequentially. For the interactive evaluation, the agent is presented with the current plan's goal along with the history of previous GT states and actions. Using this context, the agent predicts the next action and interacts step-by-step with the environment. The interaction continues until the current plan is either successfully completed or terminated due to failure (e.g., collisions or deadlocks). After each plan, the context is reset to include the GT actions and states from the completed portion of the trajectory, and the agent proceeds to the next plan. This ensures that each plan is evaluated independently, conditioned only on the correct prior history. The agent's performance is measured via cumulative reward across all plans in the trajectory. Pseudocode for this evaluation procedure is provided in Algorithm 2.

Algorithm 2 Plan-Level Evaluation

Input: Trajectory $T = \{P_1, P_2, \ldots, P_N\}$, Agent policy π, Environment $\mathcal{E}$
Initialize: Reward $R \leftarrow 0$
Initialize state and history with initial observation
for each plan P_i in T **do**
    Initialize context with GT actions up to P_{i-1}
    while not *done* and not *failure* **do**
        $a_t \leftarrow \pi(\text{context})$
        $s_{t+1}, r_t, \textit{done}, \textit{failure} \leftarrow \mathcal{E}.\text{step}(a_t)$
        Append (a_t, s_{t+1}) to context
        $R \leftarrow R + r_t$
    end while
    if *failure* **then**
        Break evaluation
    end if
end for
return Total accumulated reward R

C Additional Results

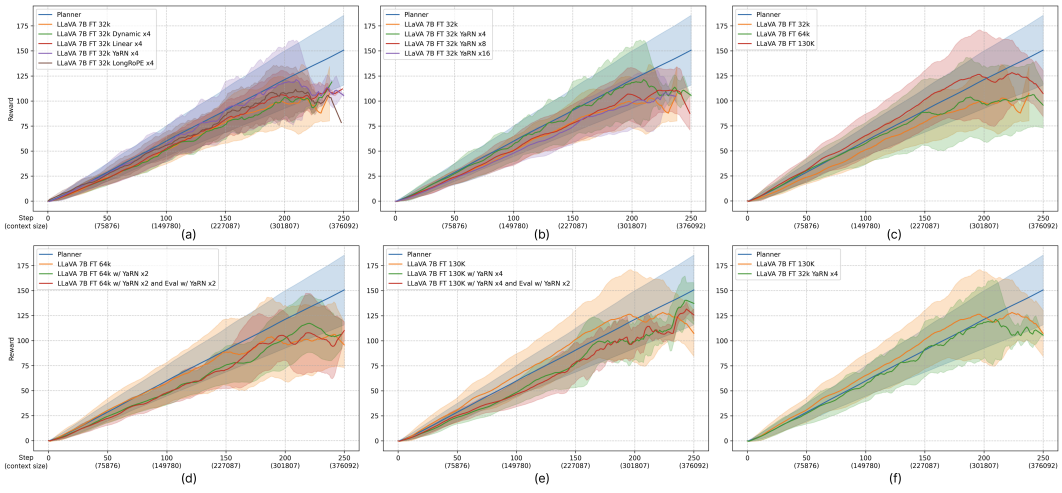

Figure 7: Agent's reward across different experimental configurations: (a) context extension methods at fixed scaling (x4), (b) varying YaRN scaling factors, (c) fine-tuning with different context lengths using Context Parallelism, (d-e) combinations of scaling during both training and inference, and (f) summary of the most effective strategies.

C.1 Interactive Evaluation: High-level Planning

Results and Discussion. Figure 7 presents the accumulated rewards over time across six experimental configurations. The Planner trajectory represents the performance upper bound. Our analysis focuses on addressing the following key questions:

Q. Which context extension methods perform best? Figure 7(a) compares different context extension methods at a fixed scaling factor of x4. Similar to the NiEH results, YaRN consistently achieves the highest performance showing very close performance to Planner.

Q. Does further scaling enhance performance? Figure 7(b) explores YaRN scaling at different scaling factors (x4, x8, and x16). Interestingly, increasing the scaling factor beyond x4 does not significantly improve performance, indicating a diminishing return for larger scaling factors.

Q. Is fine-tuning on a dataset with long trajectories effective? Figure 7(c) demonstrates the effectiveness of fine-tuning with Context Parallelism, enabling scaling of context lengths up to 64K and 130K tokens. At a 130K context size, the model can learn sequences comprising approximately 86 steps, substantially longer compared to only 22 steps with a 32K context size. This shows that exposure to longer context during training significantly enhances model performance, suggesting that incorporating more long-horizon data by ∞-THOR could further improve model capabilities. We note that context extension methods were not applied in this experiment.

Q. Does combining context extension methods during both training and inference provide additional benefits? Results of experiments with scaling at both training and evaluation (Figures 7(d) and 7(e) in Appendix) indicate that additional scaling at evaluation after fine-tuning with scaled RoPE provides no further performance improvement and may degrade performance at shorter context lengths ($\leq$300K tokens).

Based on these observations, we can conclude that fine-tuning strategies are most effective when long-trajectory datasets are available. In the absence of extensive training data, employing YaRN scaling at x4 yields performance comparable to the Planner upper-bound, particularly within context lengths under 200K tokens (Figure 7(f)).

C.2 Interactive Evaluation: Low-level Manipulation

Table 6: Success rates of OpenVLA-7B and SpatialVLA-4B on low-level Pick-up and Put tasks in ManipulaTHOR.

Task	OpenVLA-7B	SpatialVLA-4B
Pick-up	3.82%	4.68%
Put	6.38%	9.14%

We evaluate the ability of existing VLA models, OpenVLA-7B (Kim et al., 2024) and SpatialVLA-4B (Qu et al., 2025), to control robot arms on low-level manipulation tasks. The evaluation protocol follows the procedure described in Appendix B.1, with the only difference being that low-level VLA models are used to directly execute Pick-up and Put actions through arm control. For the text input, we provide task-specific instructions ("Pick-up" or "Put" object), since the existing models are only trained on single-task formulations. For the image input, we use an ego-centric camera view. Since these VLA models were originally trained on datasets with different viewing angles, there remains considerable room for improvement by incorporating additional camera perspectives during training and evaluation.

For the evaluation criterion, ManipulaTHOR implements a "magnet sphere" hand mechanism: if a pickupable object is within a specified radius of the agent's hand when the `PickupObject` action is called, the object is successfully picked up. Since neither OpenVLA nor SpatialVLA was trained in the AI2-THOR environment, we relax this success threshold by setting the radius to 0.5 to account for discrepancies between training and evaluation environments. This loosened criterion ensures that small deviations in arm trajectories do not result in an immediate failure, thereby providing a fairer comparison of the models.

Figure 6 presents accumulated rewards on low-level manipulation tasks. Both models underperform relative to the Planner baseline, largely due to differences in robot arm configuration and the out-of-distribution nature of the visual inputs. The results show that SpatialVLA achieves slightly higher and more stable rewards than OpenVLA across long sequences.

Table 6 reports task success rates. Overall performance remains low, with both models struggling to achieve success in manipulation tasks. The success rate for Put actions is slightly higher than for Pick-up, showing relatively lenient criterion for the `ReleaseObject` action compared to the `PickupObject`.

D LIMITATIONS

While ∞-THOR enables the generation of arbitrarily long trajectories, the diversity of environment layouts in AI2-THOR is inherently limited. This can lead to repetitive agent behaviors within certain scenes. For instance, compared to kitchen and living room scenes, bedroom scenes tend to involve fewer action types, mostly constrained to simple pick-and-place tasks within small spatial areas. Due to the limited scene size and low task diversity, we excluded bathroom scenes from our dev and test sets. In future work, we plan to integrate ∞-THOR with ProcTHOR (Deitke et al., 2022), which supports procedurally generated environments, enabling a broader range of dynamic and diverse scene configurations.

Additionally, although the GT action sequences generated by the PDDL-based planner are sufficient for task completion, they are not guaranteed to be optimal. This may lead to agents learning suboptimal behaviors when trained solely on these demonstrations. Incorporating learning from exploration or reinforcement-based optimization could improve policy quality.

Finally, inference with LLMs in long-context settings remains a computational bottleneck, particularly as context lengths approach 1M tokens. Since our long-horizon tasks require access to information spread throughout the entire trajectory, full-context inference becomes increasingly expensive, even with Context Parallelism. One promising direction is to equip agents with memory systems that selectively keep relevant information from prior steps, allowing the model to reason without reprocessing the full context at each step. Other architectural solutions, such as sparse attention mechanisms (Han et al., 2024; Jiang et al., 2024) or state-space models with linear-time inference (Gu & Dao, 2024), also hold potential for scalable long-context reasoning.