Where Did It Go Wrong? Capability-Oriented Failure Attribution for Vision-and-Language Navigation Agents

Jianming Chen[1,2,3,4], Yawen Wang[1,2,3,4]*, Junjie Wang[1,2,3,4]*, Xiaofei Xie[5], Shoubin Li[1,2,3,4], Qing Wang[1,2,3,4], Fanjiang Xu[1,2,3,4]*

[1]Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]Science & Technology on Integrated Information System Laboratory, Beijing, China
[3]State Key Laboratory of Complex System Modeling and Simulation Technology, Beijing, China
[4]University of Chinese Academy of Sciences, Beijing, China
[5]Singapore Management University, Singapore

{jianming2023, yawen2018, junjie, shoubin, wq, fanjiang}@iscas.ac.cn, xfxie@smu.edu.sg

Abstract

Embodied agents in safety-critical applications such as Vision-Language Navigation (VLN) rely on multiple interdependent capabilities (e.g., perception, memory, planning, decision), making failures difficult to localize and attribute. Existing testing methods are largely system-level and provide limited insight into which capability deficiencies cause task failures. We propose a capability-oriented testing approach that enables failure detection and attribution by combining (1) adaptive test case generation via seed selection and mutation, (2) capability oracles for identifying capability-specific errors, and (3) a feedback mechanism that attributes failures to capabilities and guides further test generation. Experiments show that our method discovers more failure cases and more accurately pinpoints capability-level deficiencies than state-of-the-art baselines, providing more interpretable and actionable guidance for improving embodied agents.

1 Introduction

Recently, embodied agents have garnered increasing attention for their potential to interact with various environments in applications such as navigation assistance for the visually impaired (Liu et al., 2024; Yang et al., 2022; Yuan et al., 2025) and home service robots (Shah et al., 2023; Narasimhan et al., 2025; Elnoor et al., 2025; Yan et al., 2024). These agents are designed to perform tasks that demand safety and efficiency, making their reliability essential in daily life. Any errors in their operations can lead to severe consequences, such as visually impaired users being misdirected from safe routes (Duh et al., 2021), or home service robots neglecting to turn off appliances, which could result in safety hazards (Narasimhan et al., 2025). Therefore, comprehensive testing to ensure these agents work reliably under diverse conditions is crucial.

Embodied agents integrate and rely on a combination of capabilities to perform complex tasks (Gu et al., 2022), including perception, memory, planning, and decision, with each capability serving as a component within the agent. However, due to the close interdependence of these capabilities, deficiencies in one often cascade to others, leading to compounded errors. This interplay of errors can be particularly pronounced in tasks that involve lengthy sequences, as mistakes can propagate and compound over time. A prominent illustration of this phenomenon can be found in Vision-Language Navigation (VLN) agents, which are a representative type of embodied agent designed to follow natural language instructions and navigate through environments (Wu et al., 2024). In VLN tasks, failing to associate a visual landmark with the corresponding instruction (a perception error) could lead to incorrect memory updates, flawed planning, and ultimately poor decision-making during navigation (Wang et al., 2023b, 2021; Zhu et al., 2020).

Although existing test case generation techniques are effective in identifying system-level failures, they fall short for embodied agents due to the interplay and error propagation among capabilities (Wei et al., 2022; Wang et al., 2025a; Cheng et al., 2023). Traditional methods treat agent as a monolithic entity, detecting task failures without revealing which capability caused the error or how it emerged (Chen et al., 2026). The absence of capability-oriented failure attribution restricts developers from precisely locating and resolving weaknesses, hindering targeted improvements. Therefore, there is an urgent need for a capability-oriented testing approach tailored to embodied agents—one that generates test cases enabling precise failure localization at capability level. By isolating issues within specific capabilities, it would offer greater interpretability and more actionable insights for developers.

To enable capability-oriented testing of embod-

*Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 28132–28150

ied agents, two key challenges must be addressed. **First**, it is necessary to construct capability-oriented test oracles that assess whether capability outputs are correct. Automatically building such oracles is difficult, as it requires anticipating diverse scenarios and designing distinct evaluation metrics for different capabilities. **Second**, effective failure attribution over long task trajectories is challenging. Each time step in a sequence involves a chain with multiple capabilities interacting, and these steps collectively form an even longer chain of task execution. Errors introduced by one capability often propagate and compound over time, making it difficult to trace back to the initial source of failure. Developing a mechanism to identify the initial error along this chain is essential.

In this paper, we propose a novel ***Ca***pability-orie***n***ted ***Test***ing approach, CanTest, for revealing and attributing failures to specific capabilities. Our approach includes: (1) a process for automatic generation of test cases, with the adaptive selection and mutation of case seeds, to generate test cases that are likely to expose capability-oriented failures. More notably, to address the first challenge, we introduce (2) the construction of capability oracles, which extract expected outputs and define independent evaluation metrics for each capability to determine whether errors occur. To address the second challenge, we design (3) a feedback mechanism that leverages the oracles to identify the capability responsible for a failure and computes a feedback score that considers both the failure and its source capability error. By integrating failure-oriented and capability-oriented measurements, CanTest computes feedback scores that guide the generation of test cases designed to expose both capability deficiencies and task-level failures.

To evaluate the effectiveness of CanTest, we take the VLN task as the subject of our study, and conduct experiments targeting three advanced VLN models, with results indicating that CanTest is capable of discovering the largest number of failure cases and outperforms three baselines, which are either commonly used or state-of-the-art (SOTA). The improvement in the number of discovered failure cases relative to the best-performing baseline ranges from 23.34% to 33.70%. Moreover, we conduct an experiment that plugs the capability oracles and attribution mechanisms constructed by CanTest into the baseline, expanding their ability to provide failure attribution. However, the results indicate that CanTest is still capable of discovering the largest number of failure cases targeting each capability. Next, we repair the failure case using our capability oracles, achieving repair rates ranging from 81.30% to 96.69%, thereby demonstrating the high fidelity of the oracles. Finally, through ablation experiments, we validate the significance of the feedback scores we design, showing that both failure-oriented and capability-oriented feedback contribute to discovering more failure cases. The main contributions of this work are as follows:

- To the best of our knowledge, this is the first work to propose automated test case generation specifically targeted at the individual capabilities of embodied agents.
- We automatically construct a novel set of capability-oriented test oracles designed to independently evaluate each capability of embodied agents, e.g., the perception, memory, planning, and decision-making capabilities within VLN agents.
- We designed a feedback mechanism that attributes task failures to the specific capability error and thus provides the feedback scores that comprehensively account for both failure and capability error.
- Experimental evaluations on the effectiveness of CanTest outperform all baselines, with promising performance in terms of the number of failure cases, which can all be attributed to specific capability errors[1].

2 Related Work

2.1 Testing VLN Agent

Traditionally, the quality assessment of embodied agents has relied heavily on task-specific metrics designed to quantify performance in controlled environments (Valle et al., 2025). For instance, in the realm of navigation and path planning, evaluations are often conducted using various metrics, including path length, execution time, and energy consumption (Gu et al., 2022; Wu et al., 2024).

In addition, many of these tests (Driess et al., 2023; Zitkovich et al., 2023; Wang et al., 2025b, 2024b) are primarily designed for simulation-based robots, providing various scenarios for a diverse range of tasks, such as manipulation and navigation. They often focus on scene setup and task completion without capturing the full range of capabilities required by the agents, such as reasoning quality

[1] https://github.com/JMChen121/CanTest/

and task performance quality. For instance, in the benchmarks used to evaluate PaLM-E (Driess et al., 2023) and RT-2 (Zitkovich et al., 2023), the authors emphasize the final task performance of specific agents in complex environments. VLATest (Wang et al., 2025b) is a fuzzy framework designed to generate robotic operational scenarios for testing agents, which also uses a simple oracle to evaluate the correctness of task completion.

2.2 Technology of Test Case Generation

Existing approaches primarily employ the following technologies to implement automatic test case generation: Evolutionary Algorithms (Tang et al., 2021; Zhou et al., 2023), which are inspired by biological evolution principles to optimize test case generation. For example, AVUnit (Zhou et al., 2023) supports various fuzz testing algorithms that use robustness and coverage as fitness metrics to automatically search for test cases that violate these assertions. Model-Based Search (Feng et al., 2023; Haq et al., 2022, 2023), where algorithms utilize models of the system under test to guide the search for valid test cases. For instance, SAMOTA (Haq et al., 2022) extends existing multi-objective search algorithms for test case generation, enabling efficient use of alternative models with lower running costs. Fuzzing Methods (Wei et al., 2022; Wang et al., 2025a; Cheng et al., 2023), which involve sending random or semi-random inputs to uncover vulnerabilities. For example, MoDitector (Wang et al., 2025a) can effectively generate conflict or failure scenarios and can also establish relationships between module errors and system failures.

3 Problem Statement

3.1 Vision-Language Navigation

The VLN agents aim to enable intelligent agents to perform navigation tasks in unknown environments based on natural language instructions (Zhu et al., 2020). The agent receives a natural language instruction I as input (e.g., "Walk from the kitchen to the living room, then enter the green door leading to the bedroom"). The instruction is represented as a word sequence: $I = \{w_1, w_2, w_3, \ldots, w_L\}$, where w_i denotes the i-th word in the instruction, and L is the total number of words.

The agent perceives the environment from an egocentric view and acts through viewpoint changes or low-level actions (e.g., move forward, turn left/right), producing a trajectory $\tau = s_1, \ldots, s_T$ where s_t denotes state. VLN remains challenging due to ambiguous instructions and environmental uncertainty, e.g., dynamics and noisy visual inputs (Wang et al., 2024a).

3.2 Failures of VLN Task

In the VLN task, failures of the agent are typically defined as the inability of the agent to successfully complete the given navigation task. Specifically, failures can be determined from the aspects of **target position** and **step limit**(Chen et al., 2022), as shown in Appendix A.1.

Regardless of whether the endpoint is too far from the target position or whether it has consumed additional time steps outside the limit, such situations will be deemed as task failures. In this paper, we use "*error*" to refer to the deviation between the capability output and the expected output, while "*failure*" signifies that the task has not been completed. Furthermore, it is worth noting that not all capability errors will lead to task failures.

3.3 Problem Statement

We aim to develop a capability-oriented testing approach for embodied agents. An automated test case generation mechanism is required to produce the test case set $TC = \{tc_1, tc_2, \ldots, tc_n\}$ that explores the case space comprehensively. Let a target agent $A = \{C_p, C_m, C_{pl}, C_d\}$ integrating multiple capabilities, where C_p, C_m, C_{pl}, and C_d denote the capability of perception, memory, planning, and decision, respectively. The approach should enable the construction of test oracles $\{TO_p, TO_m, TO_{pl}, TO_d\}$ for each capability $C_x \in A$ to evaluate their quality and leverage these oracles to diagnose capability-oriented errors in the trajectory τ.

Finally, for the test case tc_i, if it is a failure case, we can attribute the failure to a specific capability:

$$\begin{aligned} &(\forall \tau_i = A(tc_i), tc_i \; is \; failure) \\ &\wedge (C^{errors} = \{C_x | TO_x(C_x) = False\}) \\ &\Rightarrow \exists! C_x^* \in C^{errors}, C_x^* = Attr(\tau_i), \end{aligned} \quad (1)$$

where τ_i is produced when target agent A executes tc_i. C^{errors} denotes the set of capabilities that are erroneous (i.e., cannot be verified by the oracle). $Attr()$ represents the function, which attributes the decisive error in the τ_i to the sole capability C_x^*.

4 Approach

Our proposed CanTest contains three modules, whose overview is shown in Figure 1. First, we

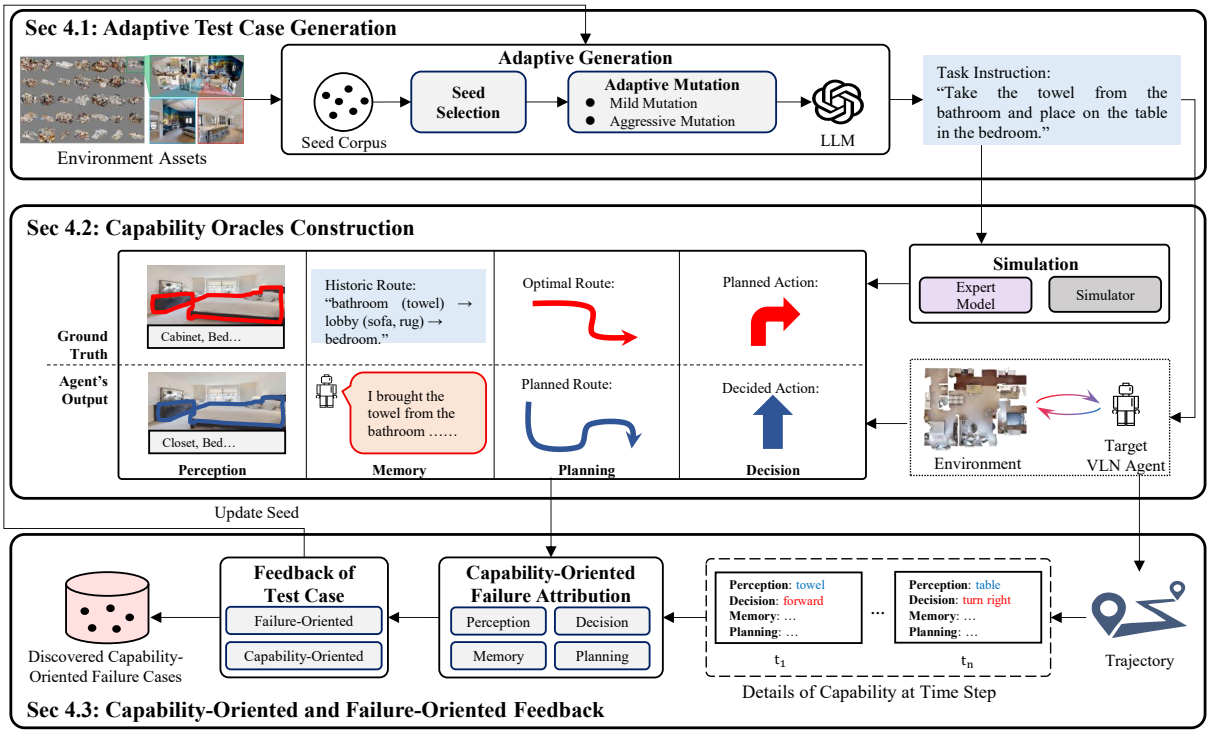


Figure 1: Overview of CanTest.

design a module of *Adaptive Test Case Generation*, which utilizes an adaptive generation mechanism to create challenging task instructions, as test cases for Vision-and-Language Navigation (VLN) tasks. The second module is *Capability Oracles Construction*. It automatically constructs oracles to determine if there are errors in the capabilities, by obtaining the expected output through prior knowledge and the execution state of the target agent and defining metrics to measure the deviation between the actual and expected output. Finally, in the *Capability-Oriented and Failure-Oriented Feedback* module, we introduce a feedback mechanism. It aims to attribute failures to specific capabilities based on oracles, thereby calculating capability-oriented feedback, which is then combined with failure-oriented feedback to guide the iterative generation of test cases.

4.1 Adaptive Test Case Generation

Following the classical search-based fuzzing methodology (Wei et al., 2022; Wang et al., 2025a), CanTest maintains a seed corpus where each seed corresponds to a test case, with a feedback score. A higher feedback score indicates a higher likelihood of uncovering capability-oriented failures. The feedback score will be further elaborated in Section 4.3. In each iteration, adaptive generation first selects seeds and then applies mutation to the selected seeds to generate new test cases.

Our scene assets are sourced from the HM3D dataset (Yadav et al., 2023), which provides a rich collection of 3D panoramic scenes, including semantic annotations of rooms and objects (denoted as A_r and A_o, respectively), forming the foundation for test case creation. Using these assets as the initial resource pool, custom-designed prompts are fed into a large language model (LLM), which combines room and object annotation to generate the task instruction: $I = \mathbb{L}(a_r, a_o)$, where $a_r \in A_r$ and $a_o \in A_o$. We treat task instructions as test cases for target agents. The prompt is shown in Appendix A.5.

CanTest first initializes a batch of test cases as the initial seed corpus SC. During each iteration of seed updates, we first calculate the selection probability for all seeds and then select a candidate based on these probabilities. The selection probability for each case seed p_{cs_i} is:

$$p_{cs_i} = \frac{max(F_{cs_i}, 0)}{\sum_{i=1}^{N} F_{cs_i}}, \tag{2}$$

where the denominator is the sum of all feedback scores. The higher the feedback score, the higher the probability of the seed being selected.

After selecting a seed s, CanTest aims to balance *failure preservation* and *search-space explo-*

ration during mutation. To this end, we design two instruction mutation operators with different intensities: *mild mutation*, which makes small semantic changes to preserve failure-relevant part, and *aggressive mutation*, which introduces larger changes to diversify trajectories and expose additional weaknesses. Moreover, CanTest adopts an adaptive strategy to select a mutation operator based on feedback score. Detailed mutation operator, selection strategy, and algorithm are provided in Appendix A.2 and A.5.

4.2 Capability Oracles Construction

Achieving capability-oriented testing requires constructing capability oracles. To address it, we propose an automated oracle-construction mechanism that derives the expected output of each capability, and then compares it with the agent's actual outputs to calculate metrics for identifying deficiencies.

We leverage expert models provided in the simulation environment to obtain the true task execution traces and related information. Specifically, the navigation expert has access to the global map and uses greedy pathfinding to produce a (near-)optimal route (Gupta et al., 2017; Kumar et al., 2018). For perception, we use the RAM image annotation model (Zhang et al., 2024) to provide expected detections and semantic annotations (e.g., object labels and bounding boxes) for the current view. When such privileged information is unavailable, similar supervision can be obtained via shortest path algorithms and manually annotated data.

Perception Oracle. We use the weighted Intersection over Union (IoU) (Rezatofighi et al., 2019) to measure the error in perception, as it is a widely recognized metric. The error is measured by:

$$\epsilon_t^p = \frac{1}{N}\left(\sum_{n=1}^{N} ||VA_{t,n} - VA_{t,n}^{gt}||_{\mathbb{L}} - \frac{|P_{t,n} \cap P_{t,n}^{gt}|}{|P_{t,n} \cup P_{t,n}^{gt}|}\right), \tag{3}$$

where t represents the t-th time step in the trajectory, N represents the number of objects detected in the current view. $VA_{t,n}$ and $VA_{t,n}^{gt}$ denote the visual annotations of the target agent and the expected output, respectively. The first term in the parentheses represents the similarity of visual annotations, and the second term corresponds to the IoU metric. The term $||VA_{t,n} - VA_{t,n}^{gt}||_{\mathbb{L}}$ is provided by the LLM, where the prompt we use is shown in Appendix A.5. $P_{t,n}$ and $P_{t,n}^{gt}$ correspond to the detected bounding boxes obtained by the target agent's perception and the expected output of perception from expert model, respectively.

Memory Oracle. We define a memory oracle that measures the accuracy of the ability to recall past information. CanTest first records the visual annotations information in the historical route and organizes them in terms of the time step, which is treated as the expected output of the memory content. Then CanTest calculates time step-oriented semantic similarity between current memory content of agent and visual annotations in the expected output to quantify errors of the memory.

$$\epsilon_t^m = 1 - ||M_t - VA_{1,\dots,t-1}^{gt}||_{\mathbb{L}}, \tag{4}$$

where M_t represents the memory description from the target agent at time step t. $VA_{1,\dots,t-1}^{gt}$ denotes the sequence of visual annotations in the historical route before time step t, and their similarity is evaluated by the LLM (restricted between 0 and 1). The designed prompt is shown in Appendix A.5.

Planning Oracle. The error of planning capability can be measured by evaluating the similarity between the trajectory generated by target agent's planned route and ground truth trajectory provided by the expert model. Since the planning process involves spatial and temporal reasoning, we adopt the normalized Dynamic Time Warping (nDTW) (Magalhaes et al., 2019) to assess the alignment between the two trajectories. nDTW is widely used for path similarity evaluation, as it accounts for both spatial proximity and temporal alignment. The error of the planning capability is measured by:

$$\epsilon_t^{pl} = 1 - \text{nDTW}(\tau_t^{pl}, \tau_{t,\dots,n}^{gt}), \tag{5}$$

where τ_t^{pl} represents route planned by target agent at time step t. $\tau_{t,\dots,n}^{gt}$ denotes the ground truth trajectory sequence from time step t onwards.

Decision Oracle. The decision commands are directly applied to the target agent and should follow the planned action provided by the upstream planning. Since the set of available actions is finite and fixed, we directly compare the differences between the actual decision actions and the planned actions in the trajectory:

$$\epsilon_t^d = 1 - ||D_t - D_t^{pl}||, \tag{6}$$

where D_t represents the action chosen by the target agent at time step t. D_t^{pl} is the planned action from the previous planned trajectory τ_{t-1}^{pl}, which serves as the expected output for the decision.

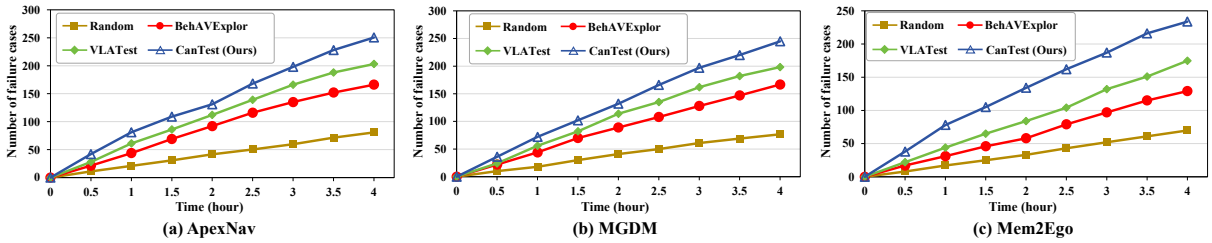


Figure 2: The comparison between CanTest and the baselines on the number of failure cases for all target models.

Table 1: Comparison of the number of failure cases toward each capability. ***+OA*** represents the integration of the capability ***O***racle and ***A***ttribution mechanism from our proposed CanTest, as a baseline plug-in.

Approach	Perception			Memory			Planning			Decision		
	ApexNav	MGDM	Mem2Ego	ApexNav	MGDM	Mem2Ego	ApexNav	MGDM	Mem2Ego	ApexNav	MGDM	Mem2Ego
Random + *OA*	20.4	22.8	21.1	16.4	15.7	11.1	19.2	18.1	16.9	24.8	20.1	21.7
BehAVExplor + *OA*	40.9	43.1	37.1	46.4	44.2	20.5	35.6	37.4	33.1	43.2	42.1	38.4
VLATest + *OA*	51.8	56.4	50.1	54.1	50.4	26.9	45.1	41.1	39.7	52.1	50.4	58.1
CanTest (Ours)	**72.2**	**74.7**	**61.4**	**66.3**	**56.1**	**42.8**	**52.5**	**49.3**	**66.1**	**59.5**	**64.7**	**63.4**

4.3 Capability-Oriented and Failure-Oriented Feedback

After obtaining the task instruction (i.e., test case) and capability oracles, CanTest lets the target agent perform the task. The resulting trajectory $\tau = \{s_1, s_2, \ldots, s_T\}$ is a time series, where the state s_t at each time step contains both the agent's own information (e.g., current position) and information about the surrounding environment (e.g., nearby objects). CanTest utilizes capability oracles to determine whether the current test case fails due to errors of the specific capability. Then, the failure-oriented and capability-oriented feedback scores are computed to simultaneously guide adaptive generation of high-value test cases. The algorithm of feedback calculation is shown in the Appendix A.3.

4.3.1 Capability-Oriented Failure Attribution

To attribute a task failure to specific capabilities, CanTest identifies which capability errors are truly responsible for the failure along a long trajectory. First, CanTest detects *capability errors* using the capability oracles (Sec. 4.2) for $\{C_p, C_m, C_{pl}, C_d\}$. Because not every detected error necessarily causes the final failure (Zhang et al., 2025c), we further determine *failure-inducing errors* via counterfactual reasoning (Chen et al., 2025a): for each detected error at time t in capability C_x, we intervene by replacing the agent output with the oracle output and roll out the remainder of the trajectory under this correction. If this intervention turns the original failed trajectory into a success, the error is deemed failure-inducing. When multiple failure-inducing errors exist, we attribute the failure to the earliest one as the *failure-source error*, and refer to its corresponding capability as the *failure-source capability*. Details of the intervention operator, indicator function, and the earliest-error rule are provided in Appendix A.4.

4.3.2 Feedback of Test Case

For the test case seed, the feedback score consists of two components: (1) *Failure-Oriented Feedback* and (2) *Capability-Oriented Feedback*. Failure-oriented feedback aims to guide the generation of the case seed that is more likely to result in failures. Capability-oriented feedback aims to guide the generation of the case seed that is more likely to result in capability-oriented failures.

(1) Failure-Oriented Feedback. Based on whether the task is successful, failure-oriented feedback F^f is defined as 0 (failure) or 1 (success).

(2) Capability-Oriented Feedback. We identify the failure-source capability and the failure-source time step, i.e., (C_x^*, t^*), as described above. For capability C_x^*, the corresponding error value $\epsilon_{t^*}^x$ is calculated by the capability oracles. The error values for each capability are normalized to the range $[0, 1]$ and are used as the capability-oriented feedback, denoted as $F^c = Norm(\epsilon_{t^*}^x)$.

The final feedback score F_{cs} of the case seed integrates both failure-oriented and capability-oriented aspects: $F_{cs} = F^f + \lambda^{C_x} F^c$. λ^{C_x} is an adaptive parameter used to control the weight of capability-oriented feedback. For failure-source capability C_x^*, if the number of failure cases caused by C_x in the already identified capability-oriented failure case set is relatively high, a lower λ^{C_x} is used to reduce the search focus on this capability.

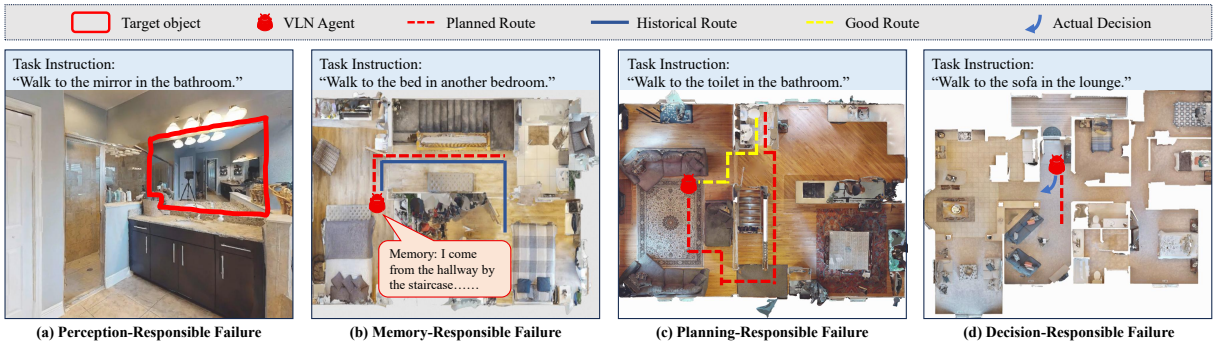


Figure 3: Examples of Failure Due to Different Capabilities.

It helps ensure comprehensive testing across all capabilities. λ^{C_x} is defined as:

$$\lambda^{C_x} = \overline{N^{C_y}}/N^{C_x},\ C_y \in \{C_p, C_m, C_{pl}, C_d\}, \tag{7}$$

where $\overline{N^{C_y}}$ denotes the average of capability-oriented failure cases in the failure case set. This score is used to guide adaptive test case generation.

5 Evaluation

Experimental Environment. We conduct experiments in Habitat3 (Gupta et al., 2017), which provides a 3D scene environment for VLN. We select three advanced VLN models as the target models to be tested: ApexNav (Zhang et al., 2025b), MGDM (Song et al., 2025), and Mem2Ego (Zhang et al., 2025a). Their details are shown in Appendix A.6.

Baselines. We compare CanTest with two commonly used techniques and one SOTA technique. (1) **Random** is an approach for randomly generating test cases, which we implement following previous work (Chen et al., 2025b; Pang et al., 2022). (2) **BehAVExplor** (Cheng et al., 2023) is a behavior-guided fuzzing technique, which is commonly used to generate diverse test cases. (3) **VLATest** (Wang et al., 2025b) is the SOTA fuzzing framework designed to generate robotic manipulation scenes for testing Vision-Language-Action (VLA) models.

5.1 Advantage of CanTest

5.1.1 Number of Failures

We present the trend of the number of discovered failure cases across the entire experimental duration of CanTest and the baselines in Figure 2. The results clearly demonstrate that, throughout each time interval, CanTest consistently outperforms the baselines. The improvement in the final number of discovered failure cases relative to the best-performing baseline (VLATest) ranges from 23.34% to 33.70%. This superior performance can be attributed primarily to the unique function of CanTest to construct oracle for various underlying capabilities. These oracles play a pivotal role in identifying the source errors in failure test cases, thereby providing more specific guidance that discovers failure cases more targeted and effective.

The feedback scores produced by this process are instrumental in guiding the iterative generation of test cases, ensuring that the focus remains on capabilities where the model struggles the most. In contrast, BehAVExplor and VLATest rely solely on feedback from system-level outcomes to direct their case generation, neglecting the capability evaluation. This limitation results in a less nuanced understanding of the specific weaknesses of the agent, hence leading to their relatively lower performance in uncovering diverse failure cases.

5.1.2 Number of Capability-Oriented Failures

Our proposed CanTest can generate failure cases that can be attributed to specific capabilities. Moreover, its components of capability oracle and attribution mechanism can be used as a plug-in (denoted as + *OA*) to extend baseline methods, thereby enabling them to perform capability-oriented failure attribution as well. Based on this, we further compare the number of capability-oriented failures identified by our proposed CanTest against those identified by the augmented baselines. Table 1 presents the results across different capabilities.

The results indicate that CanTest can discover the most failure cases for all capabilities. This reason is that CanTest promotes comprehensive exploration of different capabilities through adaptive weighting when calculating the feedback scores to guide test case generation. Specifically, across the four capabilities, CanTest significantly outperforms the baselines. This comprehensive coverage of ca-

Table 2: The results of repairing failure cases using oracles. #Fail represents the number of discovered failure test cases. #Repa and %Repa respectively represent the number and proportion of test cases successfully repaired.

Model	Perception			Memory			Planning			Decision		
	#Fail	#Repa	%Repa	#Fail	#Repa	%Repa	#Fail	#Repa	%Repa	#Fail	#Repa	%Repa
ApexNav	72.2	60.9	84.35%	66.3	53.9	81.30%	52.5	45.7	87.05%	59.5	56.6	95.13%
MGDM	74.7	62.4	83.53%	56.1	46.2	82.35%	49.3	42.6	86.41%	64.7	61.4	94.90%
Mem2Ego	61.4	52.7	85.83%	42.8	35.8	83.64%	66.1	59.3	89.71%	63.4	61.3	96.69%

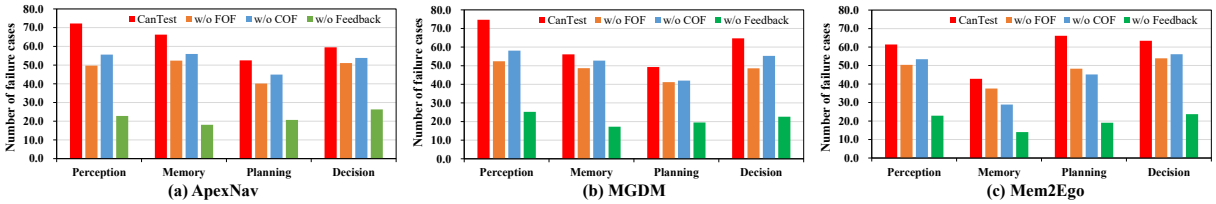


Figure 4: The results of the ablation study for feedback.

pabilities ensures that a diverse range of failures can be identified during testing, providing more helpful failure analysis for agent enhancement.

Examples of Capability-Oriented Failures. To further demonstrate the practicality of CanTest, we manually analyze the failure test cases attributed to the four capabilities it discovers, with representative examples illustrated in Figure 3. Based on these interpretable examples, we provide actionable enhancement suggestions for improving the agent. The details are presented in Appendix A.8.

5.2 Fidelity of Capability Oracles

To evaluate the fidelity of automatically constructed capability oracles by CanTest, we utilized the oracles introduced in Section 4.2 to provide expected output for each capability. For the failure cases identified by CanTest, we repaired these cases by replacing the outputs of the erroneous capabilities with the expected output. If the oracles are accurate, the test cases should be easily repaired.

The results of repairing failure cases are shown in Table 2. For each failure-responsible capability, a relatively high repair rate is achieved, i.e., repair rates ranging from 81.30% to 96.69%. This indicates that the expected output used by the oracles constructed by CanTest is reliable and accurate. Specifically, the repair rates for perception, memory, and planning, which are upstream capabilities, are lower compared to decision, which is a downstream capability. This indicates that upstream errors are often more complex, as they can propagate and impair subsequent decision execution. Across different capabilities, however, all models achieved high repair effectiveness, with repair rates exceeding 80% when using the oracles, thereby validating the effectiveness of the automatically constructed capability oracles in CanTest.

5.3 Ablation: Feedback Effectiveness

To validate the effectiveness of feedback, we compared CanTest with three variants: (1) without failure-oriented feedback, using only capability-oriented feedback (w/o FOF), (2) without capability-oriented feedback, using only failure-oriented feedback (w/o COF), and (3) without any feedback (w/o Feedback). Figure 4 displays the comparison of results between CanTest and the different variants. In all cases, CanTest outperformed the other variants, indicating that any type of feedback contributes to discovering more failure cases. Furthermore, the performance difference between w/o FOF and w/o COF is minimal, which suggests that failure-oriented and capability-oriented feedback are nearly equally important.

6 Conclusion

This paper presents CanTest, an automated framework for capability-oriented testing of embodied agents. CanTest generates capability-oriented test cases and constructs independent oracles for key capabilities, enabling systematic detection of capability errors by comparing agent outputs against expected outputs. To handle long-horizon interactions, CanTest further introduces a failure attribution mechanism to identify failure-source errors along the trajectory. Experiments demonstrate that CanTest discovers more failure cases than strong baselines, achieves the best capability-level attri-

bution performance, and attains a high repair rate, validating the fidelity of the proposed oracles. Ablation studies further confirm the effectiveness of our failure- and capability-oriented feedback signals.

Limitations

Despite its effectiveness, CanTest has several limitations that remain to be addressed in future work.
Expert Model. A key assumption in CanTest is the availability of an "expert" model to construct capability-specific oracles, e.g., (i) an optimal or near-optimal route for navigation/planning, and (ii) semantic grounding for perception (e.g., room/object labels). In settings where such an expert is not directly provided, obtaining reliable supervision may require additional prior knowledge or effort. For instance, in structured maps or partially known layouts, shortest-path planning (e.g., over a reconstructed graph) can serve as a proxy expert for near-optimal routes. Similarly, perception supervision can be approximated through extra annotation effort, such as human-labeled semantic descriptions of scenes or objects.
Simulation Environment. Our current evaluation is conducted in simulation, and directly transferring our oracle design to the physical world is nontrivial. We view CanTest as a diagnostic framework whose core idea—capability-oriented failure attribution via oracle interventions—remains applicable, but whose oracle implementations must be adapted for sim2real. In future work, we will explore (i) leveraging human-in-the-loop supervision for sparse but high-value annotations (e.g., validating key perception entities or waypoint-level plans) to calibrate oracle thresholds; and (ii) training learned surrogate oracles from real logs, where "expert" signals can be distilled from demonstrations, corrective feedback, or safety monitors. We expect that combining weak experts with uncertainty-aware oracle thresholds and selective human verification can enable CanTest to provide useful, capability-level failure attribution under a realistic environment.

Acknowledgments

This work was supported by the National Natural Science Foundation of China Grant No. 62232016, Basic Research Program of ISCAS Grant No. ISCAS-JCZD-202405 and No. ISCAS-JCZD-202304, Major Program of ISCAS Grant No. ISCAS-ZD-202401 and No. ISCAS-ZD-202302, Innovation Team 2024 ISCAS (No. 2024-66), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

Jianming Chen, Yawen Wang, Junjie Wang, Xiaofei Xie, Jun Hu, Qing Wang, and Fanjiang Xu. 2025a. Understanding individual agent importance in multi-agent system via counterfactual reasoning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(15):15785–15794.

Jianming Chen, Yawen Wang, Junjie Wang, Xiaofei Xie, Yuanzhe Hu, Qing Wang, and Fanjiang Xu. 2026. Adversarial attack on black-box multi-agent by adaptive perturbation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29359–29367.

Jianming Chen, Yawen Wang, Junjie Wang, Xiaofei Xie, Dandan Wang, Qing Wang, and Fanjiang Xu. 2025b. Demo2test: Transfer testing of agent in competitive environment with failure demonstrations. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Jinyu Chen, Chen Gao, Erli Meng, Qiong Zhang, and Si Liu. 2022. Reinforced structured state-evolution for vision-language navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 15450–15459.

Mingfei Cheng, Yuan Zhou, and Xiaofei Xie. 2023. Behavexplor: Behavior diversity guided testing for autonomous driving systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 488–500.

Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, and 3 others. 2023. Palm-e: an embodied multimodal language model. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23.

Ping-Jung Duh, Yu-Cheng Sung, Liang-Yu Fan Chiang, Yung-Ju Chang, and Kuan-Wen Chen. 2021. V-eye: A vision-based navigation system for the visually impaired. *IEEE Transactions on Multimedia*, 23:1567–1580.

Mohamed Elnoor, Kasun Weerakoon, Gershom Seneviratne, Jing Liang, Vignesh Rajagopal, and Dinesh

Manocha. 2025. Vi-lad: Vision-language attention distillation for socially-aware robot navigation in dynamic environments. *Preprint*, arXiv:2503.09820.

Shuo Feng, Haowei Sun, Xintao Yan, Haojie Zhu, Zhengxia Zou, Shengyin Shen, and Henry X Liu. 2023. Dense reinforcement learning for safety validation of autonomous vehicles. *Nature*, 615(7953):620–627.

Chen Gao, Xingyu Peng, Mi Yan, He Wang, Lirong Yang, Haibing Ren, Hongsheng Li, and Si Liu. 2023. Adaptive zone-aware hierarchical planner for vision-language navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14911–14920.

Jing Gu, Eliana Stefani, Qi Wu, Jesse Thomason, and Xin Wang. 2022. Vision-and-language navigation: A survey of tasks, methods, and future directions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7606–7623. Association for Computational Linguistics.

Saurabh Gupta, James Davidson, Sergey Levine, Rahul Sukthankar, and Jitendra Malik. 2017. Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2616–2625.

Fitash Ul Haq, Donghwan Shin, and Lionel Briand. 2022. Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th international conference on software engineering*, pages 811–822.

Fitash Ul Haq, Donghwan Shin, and Lionel C Briand. 2023. Many-objective reinforcement learning for online testing of dnn-enabled systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1814–1826. IEEE.

Ashish Kumar, Saurabh Gupta, David Fouhey, Sergey Levine, and Jitendra Malik. 2018. Visual memory for robust path following. In *Advances in Neural Information Processing Systems*, volume 31.

Shuijing Liu, Aamir Hasan, Kaiwen Hong, Runxuan Wang, Peixin Chang, Zachary Mizrachi, Justin Lin, D. Livingston McPherson, Wendy A. Rogers, and Katherine Driggs-Campbell. 2024. Dragon: A dialogue-based robot for assistive navigation with visual language grounding. *IEEE Robotics and Automation Letters*, 9(4):3712–3719.

Gabriel Ilharco Magalhaes, Vihan Jain, Alexander Ku, Eugene Ie, and Jason Baldridge. 2019. General evaluation for instruction conditioned navigation using dynamic time warping. In *NeurIPS Visually Grounded Interaction and Language (ViGIL) Workshop*, volume 1.

Siddarth Narasimhan, Aaron Hao Tan, Daniel Choi, and Goldie Nejat. 2025. Olivia-nav: An online lifelong vision language approach for mobile robot social navigation. *Preprint*, arXiv:2409.13675.

Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2022. Mdpfuzz: testing models solving markov decision processes. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 378–390. Association for Computing Machinery.

Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. 2019. Generalized intersection over union: A metric and a loss for bounding box regression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 658–666.

Dhruv Shah, Błażej Osiński, brian ichter, and Sergey Levine. 2023. Lm-nav: Robotic navigation with large pre-trained models of language, vision, and action. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pages 492–504.

Xinshuai Song, Weixing Chen, Yang Liu, Weikai Chen, Guanbin Li, and Liang Lin. 2025. Towards long-horizon vision-language navigation: Platform, benchmark and method. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 12078–12088.

Yun Tang, Yuan Zhou, Tianwei Zhang, Fenghua Wu, Yang Liu, and Gang Wang. 2021. Systematic testing of autonomous driving systems using map topology-based scenario classification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1342–1346. IEEE.

Pablo Valle, Chengjie Lu, Shaukat Ali, and Aitor Arrieta. 2025. Evaluating uncertainty and quality of visual language action-enabled robots. *arXiv preprint arXiv:2507.17049*.

Hanqing Wang, Wei Liang, Luc Van Gool, and Wenguan Wang. 2023a. Dreamwalker: Mental planning for continuous vision-language navigation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 10873–10883.

Hanqing Wang, Wenguan Wang, Wei Liang, Caiming Xiong, and Jianbing Shen. 2021. Structured scene memory for vision-language navigation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 8455–8464.

Renzhi Wang, Mingfei Cheng, Xiaofei Xie, Yuan Zhou, and Lei Ma. 2025a. Moditector: Module-directed testing for autonomous driving systems. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Ting Wang, Zongkai Wu, and Donglin Wang. 2023b. Visual perception generalization for vision-and-language navigation via meta-learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(8):5193–5199.

Zehao Wang, Minye Wu, Yixin Cao, Yubo Ma, Meiqi Chen, and Tinne Tuytelaars. 2024a. Navigating the nuances: A fine-grained evaluation of vision-language navigation. *Preprint*, arXiv:2409.17313.

Zhijie Wang, Zhehua Zhou, Jiayang Song, Yuheng Huang, Zhan Shu, and Lei Ma. 2024b. Ladev: A language-driven testing and evaluation platform for vision-language-action models in robotic manipulation. *arXiv preprint arXiv:2410.05191*.

Zhijie Wang, Zhehua Zhou, Jiayang Song, Yuheng Huang, Zhan Shu, and Lei Ma. 2025b. Vlatest: Testing and evaluating vision-language-action models for robotic manipulation. *Proc. ACM Softw. Eng.*, 2(FSE).

Anjiang Wei, Yinlin Deng, Chenyuan Yang, and Lingming Zhang. 2022. Free lunch for testing: fuzzing deep-learning libraries from open source. In *Proceedings of the 44th International Conference on Software Engineering*, page 995–1007.

Wansen Wu, Tao Chang, Xinmeng Li, Quanjun Yin, and Yue Hu. 2024. Vision-language navigation: a survey and taxonomy. *Neural Computing and Applications*, 36(7):3291–3316.

Karmesh Yadav, Ram Ramrakhya, Santhosh Kumar Ramakrishnan, Theo Gervet, John Turner, Aaron Gokaslan, Noah Maestre, Angel Xuan Chang, Dhruv Batra, Manolis Savva, and 1 others. 2023. Habitat-matterport 3d semantics dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4927–4936.

Yupei Yan, Weimin Ma, Sengfat Wong, Xuemei Yin, Qiang Pan, Zhiwen Liao, and Xiaoxin Lin. 2024. The navigation of home service robot based on deep learning and machine learning. *Journal of Robotics*, 2024(1):5928227.

Zongming Yang, Liang Yang, Liren Kong, Ailin Wei, Jesse Leaman, Johnell Brooks, and Bing Li. 2022. Seeway: Vision-language assistive navigation for the visually impaired. In *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 52–58.

Zhiqiang Yuan, Ting Zhang, Ying Deng, Jiapei Zhang, Yeshuang Zhu, Zexi Jia, Jie Zhou, and Jinchao Zhang. 2025. Walkvlm:aid visually impaired people walking by vision language model. *Preprint*, arXiv:2412.20903.

Lingfeng Zhang, Yuecheng Liu, Zhanguang Zhang, Matin Aghaei, Yaochen Hu, Hongjian Gu, Mohammad Ali Alomrani, David Gamaliel Arcos Bravo, Raika Karimi, Atia Hamidizadeh, Haoping Xu, Guowei Huang, zhanpeng zhang, Tongtong Cao, Weichao Qiu, Xingyue Quan, Jianye HAO, Yuzheng Zhuang, and Yingxue Zhang. 2025a. Mem2ego: Empowering vision-language models with global-to-ego memory for long-horizon embodied navigation. In *Workshop on Foundation Models Meet Embodied Agents at CVPR 2025*.

Mingjie Zhang, Yuheng Du, Chengkai Wu, Jinni Zhou, Zhenchao Qi, Jun Ma, and Boyu Zhou. 2025b. Apexnav: An adaptive exploration strategy for zero-shot object navigation with target-centric semantic fusion. *arXiv preprint arXiv:2504.14478*.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. 2025c. Which agent causes task failures and when? on automated failure attribution of LLM multi-agent systems. In *Forty-second International Conference on Machine Learning*.

Youcai Zhang, Xinyu Huang, Jinyu Ma, Zhaoyang Li, Zhaochuan Luo, Yanchun Xie, Yuzhuo Qin, Tong Luo, Yaqian Li, Shilong Liu, and 1 others. 2024. Recognize anything: A strong image tagging model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1724–1732.

Yuan Zhou, Yang Sun, Yun Tang, Yuqi Chen, Jun Sun, Christopher M Poskitt, Yang Liu, and Zijiang Yang. 2023. Specification-based autonomous driving system testing. *IEEE Transactions on Software Engineering*, 49(6):3391–3410.

Fengda Zhu, Yi Zhu, Xiaojun Chang, and Xiaodan Liang. 2020. Vision-language navigation with self-supervised auxiliary reasoning tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, and 1 others. 2023. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183. PMLR.

A Appendix

A.1 Background

A.1.1 Vision-Language Navigation

In this section, we take the VLN agent as the research target, which is an intelligent agent that operates in an unknown or partially known environment to perform navigation tasks based on specific instructions. Specifically, the VLN agents aim to enable intelligent agents to perform navigation tasks in unknown environments based on natural language instructions (Zhu et al., 2020). The agent receives a natural language instruction I as input (e.g., "Walk from the kitchen to the living room, then enter the green door leading to the bedroom"). The instruction is represented as a sequence of words, denoted as: $I = \{w_1, w_2, w_3, \ldots, w_L\}$, where w_i denotes the i-th word in the instruction, and L is the total number of words.

The agent perceives the environment from an egocentric (first-person) view and interacts with it through a series of discrete viewpoint changes or low-level actions (e.g., moving forward, turning left or right). The agent's execution generates a trajectory $\tau = \{s_1, s_2, \ldots, s_T\}$ formed as a time sequence, where s_t represents the agent's state at the t-th time step. The task may contain incomplete or ambiguous instructions, as well as dynamic elements and noisy visual inputs in the environment, presenting substantial challenges for navigation (Wang et al., 2024a).

A.1.2 Failures of VLN Task

In the VLN task, failures of the agent are typically defined as the inability of the agent to successfully complete the given navigation task. Specifically, failures can be determined from the following aspects (Chen et al., 2022):

Target Position: For each navigation task, after the agent executes the stop action, the task is considered successful if the distance between the agent's final position and the target position is within a certain threshold. The formula is as follows:

$$Fail_{pos} = \begin{cases} True, & \text{if } d(p_{\text{final}}, p_{\text{target}}) > \delta, \\ False, & \text{otherwise,} \end{cases} \quad (8)$$

where p_{final} denotes the agent's final position, p_{target} is the target position, $d(\cdot, \cdot)$ represents the distance function, and δ is the distance threshold.

Step Limit: If the number of steps taken by the agent exceeds the maximum step limit, the task is considered a failure. The formula is as follows:

$$Fail_{step} = \begin{cases} True, & \text{if } t > T_{\text{max}}, \\ False, & \text{otherwise,} \end{cases} \quad (9)$$

where t is the number of steps taken by the agent, and T_{max} is the maximum allowed number of steps for the task.

A.1.3 Capabilities of VLN Agent

To accomplish VLN tasks, agents usually require the following core capabilities (Gu et al., 2022; Wu et al., 2024):

- **Perception**: The agent perceives the environment from an egocentric view, extracting visual features (e.g., objects, obstacles, spatial layout) (Wang et al., 2023b). Perception directly affects the agent's ability to link observations with language instructions (e.g., identifying a "green door").
- **Memory**: VLN tasks often require short-term and long-term memory to record previously visited locations and observed scenes, helping the agent avoid revisiting the same areas (Wang et al., 2021).
- **Planning**: Planning involves designing an efficient navigation trajectory to the target, considering both language instructions and the observed environment (Wang et al., 2023a). For example, the agent may need to infer a path to a "bedroom" through an intermediate room ("living room").
- **Decision**: Decision-making determines the next action (e.g., turn left, move forward) based on real-time perception, memory, and planned trajectories (Gao et al., 2023).

These capabilities are interdependent during task execution. Perception provides the foundation for memory and planning, memory supports planning and decision by reducing redundant actions, and planning defines the trajectory that decision follows in real-time. While these capabilities collectively enable VLN agents to navigate effectively, any deficiency in these capabilities can lead to task failures.

A.2 Adaptive Test Case Generation

We design an algorithm for adaptive case generation, described in Algorithm 1. CanTest first initializes a batch of test cases as the initial seed corpus

SC (line 1). During each iteration of seed updates, we first calculate the selection probability for all seeds and then select a candidate based on these probabilities (line 2). where the denominator is the sum of all feedback scores. The higher the feedback score, the higher the probability of the seed being selected.

After selecting seed s as the candidate, we designed an adaptive mutation strategy that applies different mutation methods to candidates with varying levels of feedback scores. Essentially, mutation involves modifying the objects or rooms in the task instruction so that the agent takes a different route, to make the route more likely to reveal failures. We design prompts to apply the LLM to mutate the current instructions, and detailed prompts are available on our website. Specifically, for task instructions, we design two levels of mutation as follows.

Mild Mutation: for test cases with higher feedback scores (indicating a higher probability of failure), we apply mild mutation to the task instruction. For example, as shown in Figures 5 (a) and (b), we make slight changes without altering the destination room (i.e., modifying the destination as another object within the room).

Aggressive Mutation: conversely, for those with lower feedback scores, as shown in Figures 5 (a) and (c), using the same room as the destination may not easily lead to failures. Therefore, we apply aggressive mutation to alter the target room itself, guiding the agent along a significantly different path to expose potential weaknesses.

To adaptively select the appropriate mutation strategy, we calculate the following mutation probabilities based on the feedback score:

$$p_m = \frac{F_{cs} - min(\mathbf{F})}{max(\mathbf{F}) - min(\mathbf{F})}, \tag{10}$$

where $\mathbf{F} = \{F_{cs_1}, \ldots, F_{cs_n}\}$ and p_m represents the degree to which the feedback score of the current seed ranks among all seeds. A higher p_m indicates that current seed is more outstanding within the seed corpus, i.e., it is more likely to fail due to certain capability deficiencies. Therefore, CanTest tends to perform mild mutation to avoid excessive mutations that could lead to difficulty in revealing failure. Conversely, a lower p_m leads to a tendency for aggressive mutation, expanding the search space and striving for greater difference before and after the mutation.

A higher p_m indicates that the current seed is more outstanding within the seed library, i.e., it is more likely to fail due to certain capability deficiencies. Therefore, CanTest tends to perform mild mutation to avoid excessive mutations that could lead to difficulty in revealing failure (lines 4-8). Conversely, a lower p_m leads to a tendency for aggressive mutation, expanding the search space and striving for greater difference before and after the mutation (lines 9-13).

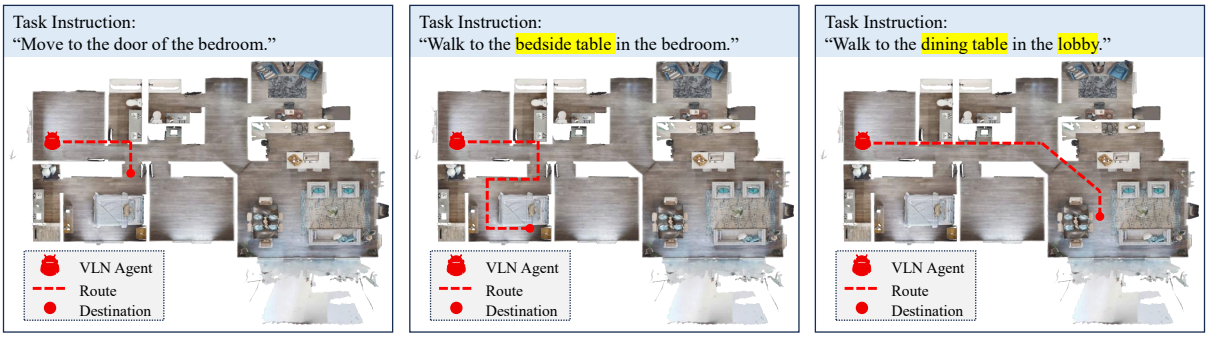
Algorithm 1: The Algorithm for Adaptive Test Case Generation.

Input: The semantic annotations A_r and A_o of environment assets.
Output: The generated test case seed s'.

1 **Initialization:** Generate the initial seed corpus SC.
2 Select candidate s from SC. with probability p_s according to the Equation 2;
3 Calculate the adaptive mutation probability p_m according to the Equation 10;
4 **if** *random()* < p_m **then**
5 Get the room annotation a_r and object annotation a_o from candidate s;
6 Sample another object: a'_o = Sample(A_o, a_r, a_o);
7 s' = MildMutation(s, a_r, a'_o);
8 **end**
9 **else**
10 Get the room annotation a_r from candidate s;
11 Sample another room and object: a'_r, a'_o = Sample(A_r, A_o, a_r);
12 s' = AggressiveMutation(s, a'_r, a'_o);
13 **end**
14 **return** s';

A.3 The Algorithm for Capability Failure Feedback

The workflow for capability and failure feedback is shown in the Algorithm 2. After obtaining the task instruction (i.e., test case) and capability oracles, CanTest lets the target agent explore the environment based on the instruction to perform the task. The resulting trajectory $\tau = \{s_1, s_2, \ldots, s_T\}$ is a time series, where the state s_t at each time step contains both the agent's own information (e.g., current position) and information about the surrounding environment (e.g., nearby objects). The various capabilities generate their respective outputs based on s_t (line 1). CanTest utilizes capability oracles to eval-



(a) Original Case **(b) New Case After Mild Mutation** **(c) New Case After Aggressive Mutation**

Figure 5: Illustration of the mild mutation and the Aggressive mutation.

uate τ, determining whether the current test case fails due to errors of the specific capability (line 2). Then, the failure-oriented and capability-oriented feedback scores are computed to simultaneously guide adaptive generation of high-value test cases.

A.4 The Details of Capability-Oriented Failure Attribution

To attribute failures to specific capabilities, it is necessary to identify the source of failures. Because capabilities interact across long task trajectories, not all capability errors necessarily lead to task failure. To address this, CanTest incorporates a failure attribution mechanism that first identifies capability errors via oracles, then determines which of these actually caused the failures (named as failure-inducing errors). Among these, the earliest occurring error is designated as the failure-source error. The capability corresponding to the error is referred to as failure-inducing or failure-source capability.

Specifically, for the capabilities of perception, memory, planning, and decision (denoted as $\{C_p, C_m, C_{pl}, C_d\}$), each capability error is evaluated by the oracle introduced in Section 4.2. We use the tuple (C_x, t) to represent an error, which means that the capability C_x ($C_x \in \{C_p, C_m, C_{pl}, C_d\}$) of the target agent at time step t produces outputs that violate the oracles. A trajectory may contain multiple errors, but not all errors necessarily lead to an overall failure (Zhang et al., 2025c). We aim to identify the truly failure-inducing errors that ultimately lead to failure within a long chain of interdependent capabilities.

CanTest attribute failures to specific capability errors, leveraging the concept of counterfactual causal reasoning (Chen et al., 2025a). Specifically, suppose the trajectory τ is a failure, i.e., $R(\tau) = Failure$, as defined in Section 3.2. Consider the following scenario: if the error in capability C_x at time t is corrected, i.e., by replacing the original output with the capability oracle. The steps before time t remain unchanged, while the actions after time t are automatically updated accordingly. This process generates a modified trajectory:

$$\tau^{(C_x,t)} = \mathbb{M}_{(C_x,t)}(TO_x, \tau), \tag{11}$$

where $\mathbb{M}_{(C_x,t)}$ represents the modification to the output of capability C_x at time step t. TO_x is the test oracle for this capability, which serves as the expected output for the capability output and is substituted into τ. If the modified trajectory $\tau^{(C_x,t)}$ results in $R(\tau^{(C_x,t)}) = Success$, then the error (C_x, t) is considered as a failure-inducing error. We define a failure-inducing error indicator as a capability-time tuple (C_x, t):

$$\delta_\tau(C_x, t) = \begin{cases} 1, & \text{if } R(\tau^{(C_x,t)}) \text{ is } Success \\ & \text{and } R(\tau) \text{ is } Failure, \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

A failure-inducing error is then a tuple (C'_x, t') with $\delta_\tau(C'_x, t') = 1$, where C'_x denotes the *failure-inducing capability* and t' represents the corresponding failure-inducing time step. Multiple failure-inducing errors may occur within a single trajectory. To handle this, we attribute the failure to the earliest failure-inducing error, defined as:

$$\begin{aligned} \mathbb{C}(\tau) &= \{(C_x, t) | \delta_\tau(C_x, t) = 1\}, \\ (C^*_x, t^*) &= \arg \min_{(C'_x,t') \in \mathbb{C}(\tau)} t, \end{aligned} \tag{13}$$

where C^*_x denotes the *failure-source capability* and t^* the corresponding failure-source time step. Accordingly, the failure of trajectory τ is attributed to the capability C^*_x at time step t^*, and τ is stored in the failure case set.

Algorithm 2: The Algorithm for Capability Failure Feedback.

Input: The generated test case seed s' and corresponding task instruction I, the oracles $\{TO_x\}$.

Output: The feedback score F_{cs} of the case seed.

Let target agent execute task according to instruction I and get trajectory τ;
Identify the capability error (C_x, t) using the $\{TO_x\}$ introduced in Section 4.2;
if *result of* τ $R(\tau) = Failure$ **then**
    Replace the (C_x, t) with the $\{TO_x\}$ to calculate indicator;
    Identify the failure-inducing error (C'_x, t') from (C_x, t) using indicator;
    Identify the failure-source error (C^*_x, t^*) from (C'_x, t');
    Store τ to the failure case set;
    Calculate the failure-oriented and capability-oriented feedback;
    Calculate the adaptive weight λ^{C_x} for capability-oriented feedback;
    return feedback score $F_{cs} = F^f + \lambda^{C_x} F^c$;
end
else
    return feedback score $F_{cs} = 0$;
end

A.5 Details of Prompts

Figures 6–10 summarize the prompts used in our framework. Specifically, we provide the templates for mild and aggressive mutation to generate candidate instructions, as well as the prompts for the perception and memory oracles that produce semantic observations and route/history information used by the planner.

A.6 Experimental Settings

A.6.1 Environment

We take the VLN agent as the subject of our study, and conduct experiments in Habitat3 (Gupta et al., 2017), a platform that provides a 3D scene environment for VLN. At each time step, the agent receives RGB observations from three directions: forward, left, and right. The agent is provided with atomic actions, including "Move Forward," "Turn Left," "Turn Right," and "Stop." When the agent executes the "Stop" action, the current task is considered complete. The scene assets used in our experiments are primarily sourced from HM3D (Yadav et al., 2023), which includes 216 large-scale indoor 3D reconstructed scenes along with their semantic annotations. We selected three advanced VLN models as the target models to be tested: ApexNav (Zhang et al., 2025b), MGDM (Song et al., 2025), and Mem2Ego (Zhang et al., 2025a).

A.6.2 Baseline

The baseline VLATest defines four operators (Target object, Confounding objects, Lighting, and Camera) and generates test cases in a fuzzing-based framework by sampling/perturbing these factors. To adapt VLATest to VLN, we keep the same operator-driven generation paradigm and make task-specific adjustments: (1) Target object becomes the VLN's destination landmark (goal-related object/room cue in the instruction); (2) Confounding objects become distractor landmarks that the agent should avoid/keep away from; (3) Lighting perturbs the perceived view via illumination-intensity changes; and (4) Camera perturbs observations via camera-rotation (viewpoint) shifts. The remaining components (e.g., object sampling strategy, perturbation magnitudes, and the fuzzing loop) follow the original VLATest framework.

A.7 Failure Diversity

To further examine whether the discovered failures are diverse rather than repeated instances of a few dominant patterns, we conduct an additional manual diversity analysis. Specifically, we design a finer-grained failure taxonomy that goes beyond the four high-level capability sources used in the main paper. The taxonomy contains eight failure types spanning perception, memory, planning, and decision-making, as summarized in Table 3.

Under this taxonomy, we randomly sample 100 failure cases on ApexNav discovered by our method and another 100 cases discovered by the best baseline, and manually assign each case to one taxonomy category. The results show that our method covers all eight categories, whereas the best baseline covers only six categories and does not discover failures of types **ME-2** (Temporal/Order Error) and **PL-2** (Looping). This suggests that our method uncovers a broader range of failure modes, instead of repeatedly generating failures from the same pattern.

System Prompt

You are a strict VLN instruction generator. Your output will be used as an automated test case. The instruction must be executable, minimally ambiguous, and grounded in the given room/object annotations. Output EXACTLY one JSON object and nothing else.

User Prompt

Create one navigation instruction I that uses the given semantic annotations.
Given:
- Destination room (a_r): "{a_r}"
- Target object (a_o): "{a_o}"
Optional additional candidates:
- Other plausible objects in the same room: {other_objects_in_room}
- Other rooms in the scene: {other_rooms}
Hard constraints (must satisfy all):
(1) The FINAL destination room must be exactly a_r.
(2) The FINAL target must be exactly a_o and must be in the destination room.
(3) 1 or 2 sentences only.
(4) No privileged info (no coordinates, no map, no "north/east", no dataset names, no "annotation").
(5) Use plain indoor navigation actions (walk/go/turn/enter/pass/stop/take/place/put).
(6) Avoid vague goals like "explore" or "look around". The instruction must end with reaching/finding/placing at a_o in a_r.
(7) If you use a two-step "take ... and place ..." structure, the second step MUST end at a_r and a_o.
Soft constraints (try to satisfy):
- Include one landmark object (not equal to a_o) if other_objects_in_room is available.
- Use deterministic phrasing (avoid "maybe", "try", "somewhere").
STYLE EXAMPLE (for style/structure only):
"Take the towel from the bathroom and place on the table in the bedroom."
Generate ONE instruction following a similar style: clear action verbs, concrete rooms/objects, and a concise two-step structure if appropriate, while strictly satisfying the hard constraints above.
Output JSON schema:
{
"instruction": "...",
"goal_room": "…",
"goal_object": "…",
"landmark_object": "... or null",
"num_sentences": 1 or 2
}
If you cannot find a landmark, set landmark_object to null.
Do not output anything outside the JSON.

Figure 6: The prompt for task instruction generation.

System Prompt

You are an instruction mutation engine for Vision-and-Language Navigation (VLN). Your goal is to minimally modify a task instruction while keeping it executable and semantically coherent. You MUST follow the constraints and output EXACTLY the JSON schema requested.

User Prompt

Perform MILD mutation of a VLN instruction.
Original instruction: "{instruction}"
Context:
- Destination room in the original instruction (if known): "{dest_room}"
- Candidate objects in the destination room (if available): {objects_in_dest_room}
- Object(s) mentioned in the original destination: {original_dest_objects}
MILD mutation definition:
- Keep the destination ROOM unchanged.
- Only modify the destination OBJECT(S) within the same room, choosing a different plausible object.
- Do NOT change the action sequence structure (e.g., "go to", "turn", "walk", "enter", "exit") except for necessary grammatical agreement.
- Preserve other constraints in the instruction (e.g., "avoid", "after", "before", "then", "until") and preserve the number of steps/clauses as much as possible.
- The mutated instruction must remain natural, concise, and unambiguous.
If objects_in_dest_room is provided, choose ONLY from that list and ensure the new object is different from original_dest_objects.
If it is not provided, choose a common household object that plausibly exists in {dest_room}.
Output format:
Return exactly one JSON object with:
{
"mutation_type": "mild",
"original_instruction": "...",
"mutated_instruction": "...",
"changed_spans": [
{"from": "...", "to": "...", "reason": "..."}
]
}
Do not output any explanations outside the JSON.

Figure 7: The prompt for mild mutation.

System Prompt

You are an instruction mutation engine for Vision-and-Language Navigation (VLN). Your goal is to substantially modify a task instruction to force a different navigation route while keeping it executable and semantically coherent. You MUST follow the constraints and output EXACTLY the JSON schema requested.

User Prompt

Perform AGGRESSIVE mutation of a VLN instruction.
Original instruction: "{instruction}"
Context:
- Original destination room (if known): "{dest_room}"
- Candidate alternative destination rooms (if available): {candidate_rooms}
- Allowed room types (if available): {room_types}
- Known start area / current area (optional): "{start_area}"
AGGRESSIVE mutation definition:
- Change the destination ROOM to a different plausible room (not equal to the original destination room).
- Update the destination OBJECT(S) accordingly so they are plausible in the new room.
- Keep the instruction's overall intent and structure (number of clauses, ordering words like "then/after/before") as much as possible, but rewrite the minimal necessary parts to ensure coherence.
- Do NOT introduce impossible requirements (e.g., two destinations that conflict).
- Preserve constraints unrelated to the destination when possible (e.g., "avoid the stairs", "do not enter the bedroom", etc.). If a preserved constraint would contradict the new destination, adjust it minimally and explicitly record it in changed_spans.
If candidate_rooms is provided, choose ONLY from that list and ensure it differs from dest_room.
If not provided, choose a common household room that is different from dest_room (e.g., kitchen, bathroom, bedroom, living room, hallway, dining room, office).
Output format:
Return exactly one JSON object with:

```
{
  "mutation_type": "aggressive",
  "original_instruction": "...",
  "mutated_instruction": "...",
  "changed_spans": [
    {"from": "...", "to": "...", "reason": "..."}
  ],
  "new_destination_room": "...",
  "new_destination_object": "..."
}
```

Do not output any explanations outside the JSON.

Figure 8: The prompt for aggressive mutation.

System Prompt

You are a strict semantic similarity evaluator for vision-language annotations. Your job is to output a single numeric distance d in [0, 1] measuring how different two annotations are. d = 0 means semantically equivalent; d = 1 means completely unrelated or contradictory. Be consistent and deterministic. Do not output anything except the JSON specified.

User Prompt

Compute the semantic distance between the agent annotation and the expected annotation.
Agent annotation (VA): "{VA}"
Expected annotation (VA_gt): "{VA_gt}"
Scoring rules (distance d in [0,1]):
- d = 0.00: identical meaning or exact synonyms (e.g., "sofa" vs "couch").
- d = 0.10: same object/category with minor wording differences (e.g., "dining table" vs "table"; "wooden chair" vs "chair").
- d = 0.30: same broad category but notable mismatch in specificity/attributes that could matter (e.g., "desk" vs "table"; "armchair" vs "chair").
- d = 0.60: related but different categories or plausible confusion (e.g., "cabinet" vs "refrigerator"; "lamp" vs "ceiling light").
- d = 0.90: unrelated categories (e.g., "bed" vs "microwave").
- d = 1.00: explicit contradiction (e.g., "door" vs "window" when clearly distinct) or mutually exclusive labels.
Additional guidance:
- Ignore capitalization, punctuation, and plurality.
- Treat common hypernym/hyponym relations as close (table vs dining table) but not identical.
- If either annotation is empty/unknown ("unknown", "none", ""), set d = 0.80 unless both are unknown/empty, in which case d = 0.00.
Output format:
Return exactly one JSON object: {"d": <number>} with <number> rounded to 2 decimal places.
Do not include explanations.
Examples:
1) VA="sofa", VA_gt="couch" -> {"d": 0.00}
2) VA="table", VA_gt="dining table" -> {"d": 0.10}
3) VA="bed", VA_gt="microwave" -> {"d": 0.90}

Figure 9: The prompt for perception oracle.

System Prompt

You are a strict semantic similarity evaluator for embodied-agent memory. Your job is to output a single numeric similarity s in [0, 1] measuring how well the agent's memory description matches the ground-truth historical visual annotations up to time t-1. s = 1 means the memory is fully consistent and accurately reflects the history; s = 0 means it is unrelated or contradictory. Be consistent and deterministic. Do not output anything except the JSON specified.

User Prompt

Compute the semantic similarity between the agent's memory description at time t and the historical visual annotations along the ground-truth route before time t.
Agent memory (M_t): "{M_t}"
Ground-truth history (VA_gt_{1..t-1}): {VA_gt_list}
Notes:
- VA_gt_{1..t-1} is a time-ordered sequence. Each element corresponds to a past time step and contains the visual annotations (e.g., detected objects/rooms/attributes).
- The memory M_t may be a compressed summary; it does NOT need to mention every item to get a high score.
Scoring rules (similarity s in [0,1]):
- s = 1.00: Memory is fully consistent with the history; it correctly mentions the key visited rooms/objects/events (allowing synonyms) and contains no hallucinated entities/events that are absent from the history. If the memory includes temporal/order statements, they align with the sequence.
- s = 0.80: Mostly consistent; minor omissions of less important details are allowed; at most negligible extra details that do not change the meaning.
- s = 0.50: Partially consistent; mentions some correct history but misses many key elements OR includes some hallucinated/incorrect claims that affect correctness.
- s = 0.20: Largely inconsistent; only weak overlap with history OR many hallucinations/contradictions.
- s = 0.00: Completely unrelated to the history or directly contradicts it.
Evaluation guidance:
- Treat synonyms/hypernyms as matches when reasonable (e.g., "sofa"~"couch", "dining table"~"table").
- Ignore capitalization, punctuation, and plurality.
- Focus on factual consistency with VA_gt_{1..t-1}:
  * Penalize hallucinations (entities/rooms/actions not supported by history) and contradictions.
  * Do NOT penalize concise summaries that omit minor details.
- Time-step consistency: only evaluate ordering if M_t explicitly references order (e.g., "first..., then..."). If M_t has no explicit order cues, do not penalize for missing ordering.
Output format:
Return exactly one JSON object: {"s": <number>} with <number> rounded to 2 decimal places.
Do not include explanations.
Examples:
1) M_t="I went from the kitchen to the living room and saw a sofa."
   VA_gt includes kitchen then living room with sofa -> {"s": 0.90}
2) M_t="I was in the bedroom and saw a microwave."
   VA_gt contains only hallway and bathroom with sink -> {"s": 0.00}
3) M_t="I passed through a hallway and ended near a table."
   VA_gt contains hallway and dining table (but many other objects) -> {"s": 0.70}

Figure 10: The prompt for memory oracle.

Table 3: Finer-grained failure taxonomy used in the manual diversity analysis. Each failure type is defined by its characteristic evidence in the trajectory and its typical consequence.

Capability	Failure Type	Decision Evidence	Consequence
Perception	PE-1 Missed Detection	At step t, the target object or landmark is visible but not perceived.	Misses a critical entrance or landmark.
Perception	PE-2 Object Confusion	The target is perceived as a similar but different object or landmark.	Moves toward a similar but incorrect room or landmark.
Memory	ME-1 Forgetting	Previously visited paths or landmarks are not retained in memory.	Leads to redundant search or backtracking.
Memory	ME-2 Temporal/Order Error	The visitation order of landmarks is confused or misassigned.	Violates ordering constraints such as *before*, *after*, or *pass-after*.
Planning	PL-1 Detour	Fails to choose a shorter route and instead follows a longer one.	Takes a long detour and reaches the step limit.
Planning	PL-2 Looping	Repeatedly passes the same landmarks or revisits the same area.	Falls into a loop until the step limit is reached.
Planning	PL-3 Constraint Violation	Fails to follow the instruction constraints.	The route violates constraints such as pass/avoid/order requirements.
Decision	DE-1 Inconsistent with Plan	The executed action deviates from the previously planned next step.	Makes an incorrect turn, e.g., at an intersection.

A.8 Usefulness of Capability-Oriented Failures

1. Perception-Oriented: as shown in Figure 3 (a), the agent is already positioned in front of the mirror as required by the instructions. However, due to insufficient perception capabilities, it cannot recognize the mirror, thus resulting in the agent moving away from the target object and being unable to complete the task instructions in the subsequent trajectory.

   Enhancement Suggestion: Improve the understanding of the environment by providing more context about the surrounding environment. For example, by integrating perception results from adjacent time steps, the Spatiotemporal continuity of context can be leveraged to reduce perception uncertainty.

2. Memory-Oriented: as presented in Figure 3 (b), the instruction requires the agent to walk to another bedroom. However, after the agent traveled from one bedroom to another, it lost the long-term memory of originally coming from the first bedroom. This deficiency in memory subsequently cause the agent to leave the current bedroom again, leading to task failure.

   Enhancement Suggestion: Consider introducing a better integration mechanism of short-term and long-term memory, allowing the agent to retain necessary information while switching between different rooms.

3. Planning-Oriented: as illustrated in Figure 3 (c), the agent has different routes to reach the location specified in the task instructions. Due to poor planning, the agent did not choose a better route. The planned route is too long,

ultimately causing the time steps to exceed the limit and resulting in failure.

Enhancement Suggestion: Improve the agent's path planning algorithms by allowing real-time analysis of the environment and dynamic adjustments to choose better routes. Additionally, set reasonable time budgets to ensure the agent considers time constraints during the planning process.

4. Decision-Oriented: as shown in Figure 3 (d), the agent does not follow the planned route for decision-making. Due to its unwarranted autonomy in decision-making, it deviates from the planned route, ultimately leading to an improper decision that results in the inability to complete the task.

   Enhancement Suggestion: Limit the agent's autonomy in decision-making, and promptly assess whether adjustments are needed when decisions are inconsistent with the planned route.