Hierarchical Planning Agent for Web-Browsing Tasks

Anonymous Author(s)
Affiliation
Address
email

Abstract

Recent advances in large language models (LLMs) have enabled the development of agentic systems for sequential decision-making. Such agents must perceive their environment, reason over multiple time steps, and take actions that optimize long-term goals. However, existing web agents perform poorly on complex long-horizon tasks due to key limitations: limited in-context memory for tracking history, weak planning that fails to satisfy user constraints, difficulty handling task complexity, and greedy behaviors that cause premature termination. To address these challenges, we propose Structured Agent, a hierarchical planning framework with two core components: (1) an online hierarchical planning algorithm that uses dynamic AND/OR trees for efficient search, and (2) a structured memory module that tracks candidate solutions to improve constraint satisfaction in information-seeking tasks. Experiments on WebVoyager and custom shopping benchmarks demonstrate that Structured Agent achieves improvements in long-horizon reasoning and planning compared to standard LLM-based agents.

1 Introduction

Large language models (LLMs) have recently demonstrated remarkable performance across a wide range of natural language processing tasks. This progress has led to the emergence of *agentic LLMs*, which augment language models with the ability to perceive, reason, and act within interactive environments. Agentic LLMs are now deployed in diverse applications, including enterprise workflow automation, autonomous research assistance, customer support, and software development.

A particularly promising application lies in the domain of *web agents*, which browse and interact with the internet to perform complex, goal-driven tasks such as information seeking, form filling, transactional activities, and monitoring. However, this area remains relatively underdeveloped, with many open challenges in building robust, and interpretable web-based agents.

Despite recent advances, current web agents continue to struggle on *long-horizon, multi-step web-browsing tasks* [Yang et al., 2025, Erdogan et al., 2025, Qi et al., 2025]. Even when powered by strong LLMs such as GPT-4o, studies show that agents frequently fail on tasks requiring fewer than 15 steps [Erdogan et al., 2025, Qi et al., 2025]. These shortcomings arise from weak task planning, limited context memory, insufficient error recovery mechanisms [Koh et al., 2025], and greedy decision-making that leads to premature commitment to suboptimal strategies [Schmied et al., 2025](See appendix B). Open-source agents are especially vulnerable, as they often lack the contextual capacity and domain expertise of proprietary models [Murty et al., 2025].

To address these challenges, recent efforts [Qi et al., 2025, Murty et al., 2025] have explored curriculum-based reinforcement learning (RL) using expert- or LLM-generated trajectories to fine-tune open-source models. While this improves performance, it is constrained by the high cost and limited scalability of trajectory collection and does not fully exploit the *compositional structure* inherent in many real-world tasks.

Submitted to NeurIPS 2025 Workshop on Efficient Reasoning. Do not distribute.

A promising alternative is to equip agents with explicit planning capabilities. Inspired by human problem-solving, hierarchical planning allows agents to reason over tasks at multiple levels of abstraction by recursively decomposing complex goals into structured hierarchies of subgoals and actions. This approach improves efficiency, enhances robustness through better error isolation and recovery, and increases the interpretability of the agent's behavior. Recent works have tried to integrate explicit planning mechanisms into web agents. Koh et al. [Koh et al., 2025] explored tree-based planning with A* search, Yang et al. [Yang et al., 2025] investigated incremental tree construction using prune-and-branch strategies, and Erdogan et al. [Erdogan et al., 2025] introduced planning with dynamic revisions. While these approaches advance agents beyond purely reactive behavior, they primarily operate at the level of individual actions and therefore fail to exploit the compositional structure of complex tasks, limiting overall planning efficiency.

To address the limitations of prior approaches, we introduce a hierarchical planning framework that enables agents to dynamically construct and execute ordered AND/OR planning trees at inference time, thus interleaving planning with execution. This hierarchical structure integrates AND nodes, representing mandatory subgoals, and OR nodes, representing alternative sub-strategies. By combining these, agents can decompose complex tasks into subgoals, reason about fallback options, and produce interpretable, modular plans that transparently reflect their internal decision-making process. Our framework employs a greedy depth-first search strategy for expansion, coupled with dynamic pruning and plan revision mechanisms. These features enable agents to efficiently revise or prune parts of the plan as new information becomes available and to explore large, complex search spaces more effectively. Furthermore, for information-seeking tasks in web-browsing environments, where agents are required to identify or recommend entities that fulfill user-specified constraints, we incorporate a structured memory module that tracks potential entities and the constraints they satisfy, thereby enhancing constraint satisfaction.

Our main contributions are as follows:

- We propose a hierarchical planning framework that enables inference-time planning through the use of ordered AND/OR trees. This framework facilitates compositional reasoning, dynamic pruning, and flexible plan revision, supporting more robust and adaptive decision-making.
- For web-based information-seeking tasks, we introduce a structured memory module that tracks potential entities and the constraints they satisfy, improving constraint satisfaction during planning and execution.
- We conduct experiments on the challenging WebVoyager benchmark and a custom-designed shopping benchmark, demonstrating the efficacy of our proposed method.

2 Related Works

Prior work on web agents can be broadly be classified into two main categories: (1) LLM inference-based web agents and (2) fine-tuned LLMs as web agents.

LLM Inference-Based Web Agents: Several prior works [Putta et al., 2025, Zhang et al., 2024, He et al., 2024, Zhou et al., 2024] have leveraged closed-source large language or multimodal models for complex web-based tasks, often by employing inference-time strategies. For example, Wang et al. [2025] use LLMs to extract patterns from examples and prior trajectories to guide navigation and decision-making. Other approaches incorporate Monte Carlo Tree Search (MCTS) [Zhang et al., 2024] or value function networks [Koh et al., 2024] to facilitate exploration and backtracking during planning. Methods such as Reflexion [Shinn et al., 2023] add reflection mechanisms and evaluators, though often at the cost of frequent resets.

Hierarchical planning has also been investigated as a means of addressing compositional web tasks. Erdogan et al. [2025] introduce an optimization layer for higher-level planning, while Sodhi et al. [2024] propose dynamic multi-level control, though their approach depends on domain-specific prompt engineering and task-specific policies. Yang et al. [2025] present a lightweight tree structure with effective text filtering and incremental plan revision, which outperforms many baselines but struggles on more complex tasks due to limited error recovery and weak information retention. In contrast, our method develops a more general AND/OR planning tree that supports robust error

handling and strategic planning with explicit dependency tracking, yielding more interpretable trajectories (see Table 3). We discuss the rest of the relevant works in Appendix C.

3 Preliminaries

Web-browsing tasks are partially observable and inherently stochastic. This arises because the content and structure of web pages can change dynamically over time, often unpredictably or in response to agent's actions on the web-browser, and in most cases the agent does not have access to the full underlying state of the web-browser environment. Typically, the agent is limited to information about the current HTML webpage, its prior actions, and the extracted metadata. As a result, the agent is required to make decisions under uncertainty while maintaining internal representations of its beliefs about the environment.

To capture this uncertainty and partial observability, we model the interaction between the agent and the web browser as a Partially Observable Markov Decision Process (POMDP) $\langle \mathcal{O}, \mathcal{S}, \mathcal{A}, \mathbb{R}, \mathbb{P}, p_0, \gamma \rangle$. Here, $o \in \mathcal{O}$ denotes the observations available to the agent, including the task description, plans constructed by the agent, the HTML DOM of the web page, and additional metadata from its history of interactions with the browser. The variable $s \in \mathcal{S}$ represents the true underlying state of the web browser environment and the agent. An action $a \in \mathcal{A}$ may be either an atomic browser operation (such as click, scroll, type, or go back) or a sequence of tokens describing a sub-goal or strategy. The reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ assigns a real-valued reward to each state-action pair (s, a), while the transition function $T : \mathcal{S} \times \mathcal{A} \to \Delta^{|\mathcal{S}|}$ specifies a probability distribution over possible next states for each state-action pair. The term $p_0 \in \Delta^{|\mathcal{S}|}$ denotes the initial state distribution, and $\gamma \in (0, 1)$ is the discount factor. We represent the agent's policy as a function $\pi : \mathcal{S} \to \Delta^{|\mathcal{A}|}$ mapping states to distributions over actions. Importantly, the optimal policy in a POMDP is generally history-dependent.

We use a LLM to represent the agent's policy. At each time step t, let h_t denote the history of observations, where $h_t = \{o_1, o_2, \ldots, o_t\}$. This history is encoded as a sequence of tokens $x_t = [x_1, x_2, \ldots, x_m]$ that serves as the input prompt to the LLM. The action a_t is generated as a sequence of output tokens $a_t = [y_1, y_2, \ldots, y_n]$ produced by the LLM. The probability of selecting action a_t given the history h_t is given by $\Pr(a_t \mid h_t) = \prod_{i=1}^{n} \pi(y_i \mid x_1, \ldots, x_m, y_1, \ldots, y_{i-1})$.

While this approach allows the LLM to utilize past observations, robust performance in web-browsing tasks often hinges on the ability to look ahead, reason about future contingencies, and adjust its strategy as new information is revealed. To address this, we propose a hierarchical framework that interleaves planning and execution: at each decision point, the agent uses its history of observations and actions as context, invoking the LLM policy to either revise its structured plan or sample the next action for execution as appropriate. This approach enables the agent to dynamically adapt its plans in response to the stochastic nature of the environment and its own actions.

Following prior work, we use the WebArena simulator to evaluate our agent on web-browsing tasks. WebArena exposes a set of mouse and keyboard operations as discrete actions available to the agent, including `type`, `scroll`, `click`, `go_back`, and `go_home`. Please see Appendix A of Yang et al. [2025] for detailed descriptions of these actions.

We now introduce the *Structured Agent* framework, which consists of two core components: a *Structured Memory* module that efficiently tracks intermediate decisions and candidate solutions during exploration, and an *AND/OR Tree Planner* that hierarchically decomposes tasks while reasoning over alternative strategies. Together, these components enable robust error isolation, dynamic replanning, and more reliable execution in complex web environments.

4 Structured Agent Framework

Our Structured Agent tightly interleaves planning and execution to solve complex, open-ended web-based tasks. Central to our approach is an explicit AND/OR planning tree, which the agent constructs and maintains dynamically. This tree enables hierarchical task decomposition, explicit reasoning over alternatives, and real-time plan adaptation as new observations arrive. Our agent incrementally expands and revises the planning tree, including pruning unpromising paths, in response to new

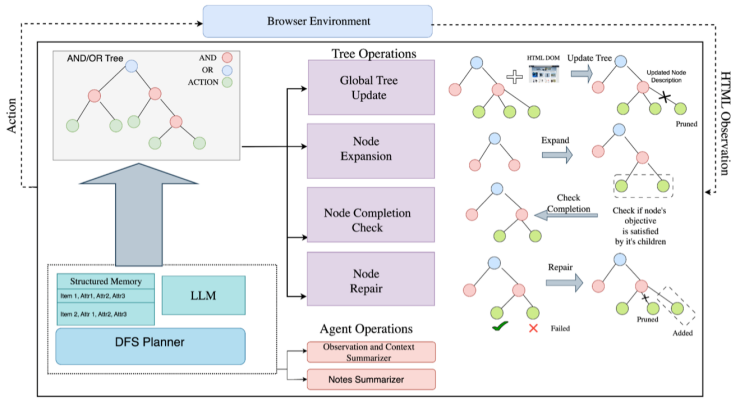

Figure 1: High-level overview of Structured Agent Framework

information. This dynamic approach significantly improves computational efficiency and prevents the planning tree from growing excessively large.

Conventional LLM agents cannot autonomously build and execute complex hierarchical plans. Rather than relying on the LLM to generate or manage the full planning tree as done by AgentOccam [Yang et al., 2025], our framework handles tree construction and maintenance. We delegate only targeted, well-scoped operations such as node expansion, repair, completion checking, and pruning to LLM calls. This clear separation of responsibilities improves both reliability and interpretability of the agent's behavior. Detailed descriptions of these operations appear in section 4.2.

We begin by introducing the formal structure of the AND/OR planning tree, and and then describe the node-state-tracking mechanisms. Then, we present the four core tree operations and two observation aggregation modules that monitor task progress and agent state. We conclude this section by describing the overall planning algorithm that integrates these components for efficient and adaptive execution.

4.1 AND/OR Planning Tree Structure

The foundation of our planning framework is an explicit AND/OR planning tree that guides both the agent's reasoning and its actions. Within this tree, the agent distinguishes between three types of nodes. (1) *AND* nodes represent conjunctive subgoals, requiring all children to succeed before the parent node can be marked as successful. Depending on the task, the ordering of AND node children may or may not matter. (2) *OR* nodes represent alternative strategies for achieving a subgoal, if any child of an OR node succeeds, the parent node is also considered successful. (3) *ACTION* nodes serve as atomic leaves in the tree. Each ACTION node corresponds to a specific browser-level operation, such as typing a query, scrolling, clicking a button, navigating back or home, or recording a note.

This hierarchical structure allows the agent to decompose tasks into flexible subgoals while explicitly reasoning about alternative solution paths. For example, when solving a task such as "`find a coffee filter under $50`", the agent may construct an AND node with two children corresponding to searching for "coffee filter" and applying a price filter. Each subgoal may further be decomposed or branched into OR nodes that represent different navigation or filtering strategies.

We allow both AND and OR nodes to expand into any combination of AND, OR, or ACTION nodes, enabling general task decompositions while capturing the sequential dependencies that are common in web-based workflows.

4.1.1 Node State Tracking

Our framework dynamically constructs an AND/OR planning tree, leveraging a LLM as a controller and expanding nodes using a greedy depth-first search (DFS) strategy. Nodes are explored in depth-first order: for AND nodes, children are executed sequentially as specified; for OR nodes, the agent greedily selects which child to execute based on the LLM's estimated likelihood of success,

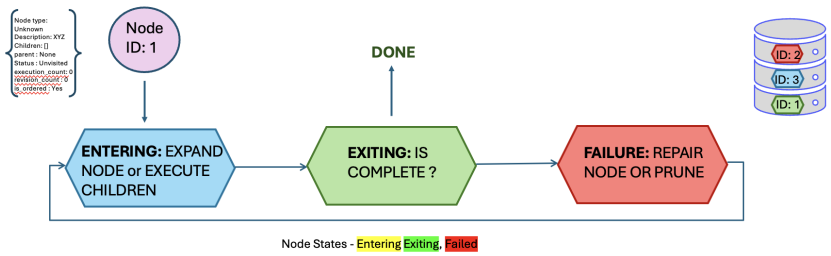

Figure 2: Node state transitions during iterative modified greedy depth-first search in an AND/OR tree. Each node may be processed multiple times and can enter the **Entering** and **Failure** states repeatedly. However, once a node reaches the **Exiting** state successfully, it is considered complete and will not be revisited.

Item / Bundle	Price ($)	Native 1080p	Bluetooth ≥5.0	Brightness (≥300 ANSI)	Includes Screen (70–100")	Status	Notes
Projector A	420	✅	✅	350	❌	🔍 To Explore	Need compatible screen under $30
Projector B + Screen Bundle	460	✅	✅	400	✅	❌ Deleted	Over budget
Projector C	390	❌	✅	500	❌	❌ Deleted	No native 1080p
Projector A + 80" Screen	445	✅	✅	350	✅	✅ Candidate	Meets all constraints

Figure 3: Example of Structured Memory for tracking constraints satisfaction of candidate solutions in information-seeking tasks

conditioned on summaries of past actions and observations. Alternatively, action selection can be guided by a learned value function, as in Koh et al. [2024].

Unlike classical DFS, our search space is stochastic, reflecting both environmental uncertainty and the agent's own actions. The agent can also dynamically prune or revise the tree, which may require nodes to be revisited multiple times. When a node is pruned or revised, these changes must be propagated upward to its parent and relevant ancestors to maintain global plan consistency. Notably, if a child of an AND node is pruned or fails, any subsequent siblings in the execution order become inapplicable and can no longer be executed. These requirements motivate several key modifications to standard DFS.

Classical depth-first search marks nodes as either *UNVISITED* or *VISITED*, but this binary distinction is insufficient for dynamic pruning and revision. Instead, we assign each node one of six possible statuses: *UNVISITED*, *VISITED*, *SUCCESS*, *FAIL*, *PRUNED*, or *DELETED*. Every node begins as *UNVISITED* and is marked *VISITED* upon processing. A node is marked *SUCCESS* if its objective is achieved; for AND nodes, this means all children succeed and collectively satisfy the node's objective; for OR nodes, any single child's success suffices; for ACTION nodes, the corresponding browser operation must succeed.

Nodes are marked *FAIL* when their objectives cannot be achieved. For AND nodes, this occurs if any child fails or is pruned; for OR nodes, if the selected child fails; for ACTION nodes, if the operation is invalid or unsuccessful. Nodes are labeled *PRUNED* if they become irrelevant or irresolvable due to new information. If the failure or pruning of a child in an AND node renders subsequent siblings inapplicable (due to ordering constraints), those siblings are marked *DELETED*. This fine-grained status labeling enables accurate tracking of execution, principled backtracking, and robust recovery from errors or dead ends.

To support upward propagation of changes when a node is pruned or revised, we introduce three additional execution states for each node on the DFS stack: *ENTERING*, *EXITING*, and *FAILED*.

A node is pushed onto the stack in the *ENTERING* state when it has not yet been processed or it still has unprocessed children. It transitions to the *EXITING* state if it has been executed (in the case of an ACTION node), once all of its children have been processed (for an AND node), or after the selected child has been processed (for an OR node), and it is time to validate whether the node's objective has been satisfied. A node enters the *FAILED* state if execution fails (for ACTION nodes) or if any of its

children are pruned or fail, indicating that the node must be repaired or pruned. Notably, a node may enter the stack in the *ENTERING*, *EXITING*, or *FAILED* state multiple times.

4.2 Tree Operations and Agent Modules

Recall that the agent's policy is generally dependent on the history of observations. In practice, it may not be feasible to contain the entire history of observations within the LLM's in-context memory. Following prior works, we generate summaries of the interactions with the browser to use as substitutes for the full history of observations.

To organize and utilize these summaries effectively, the agent maintains several internal representations. The *interaction history* logs all observation summaries and actions up to the current time step, while the *task progress summary* tracks progress toward the main objective. The *notes summary* aggregates information from all past observations that are relevant to the task, and the *task constraints* encode global and item-level requirements from the task description. At each step, the *observation summary* distills salient information from the current web page, and the *action history* lists all executed browser actions. This metadata is supplied to all planning and execution operations, enabling the agent to reason about its current estimate of the environment's state. By explicitly managing this metadata, the agent overcomes the limited context window of LLMs and maintains a coherent global view of the task throughout execution.

Using this metadata, our framework implements four core tree operations to dynamically expand, prune, and maintain the planning tree: Node Expansion, Node Repair, Global Tree Update, and Node Completion Check. Each operation involves context-specific inference from the backbone LLM, which is subsequently verified and executed by our framework. Full details of the LLM prompts, inputs, and outputs for each operation are provided in the Appendix.

The *Node Expansion* operator determines the type of a node and, if applicable, generates its children. For AND nodes, this operation outputs subgoal descriptions, for ACTION nodes, it produces a browser-level command, for OR nodes, it returns candidate strategies and associated scores. Upon receiving this output, the agent updates the tree structure. If expansion fails, the agent retries up to a preset limit, persistent failure results in the node being marked as *FAILED*.

The *Node Repair* operator handles the revision of failed AND or OR nodes by adjusting their children. It consumes the node's description, the status of its children, relevant metadata (including the current observation, sibling and parent statuses, and summaries of prior observations and notes), and generates instructions for pruning or adding children. The agent validates and enacts these instructions, if node repair fails after several attempts, it prunes the node.

The *Global Tree Update* module processes new observations to revise the tree structure. Given the latest observation, the current (pruned) tree, the task progress summary, and the notes summary, the module outputs instructions for pruning irrelevant subtrees and updating node descriptions. This module aggressively eliminates unpromising branches. As with other operations, repeated failure triggers a fallback strategy or additional pruning.

The *Node Completion Checker* operator determines whether an AND node has met its objective, based on the statuses of its children. We invoke this module when all children have completed and at least one has succeeded. It evaluates the node's description, the task progress summary, notes summary, and task constraints, returning a binary verdict of completion.

In addition to these tree operations, the agent employs two state aggregation modules to manage its internal representations and summarize progress. The *Observation and Context Summarizer* processes new observations to maintain an up-to-date summary of the environment and guide the agent's next actions. This module takes as input the current observation, task progress summary, notes summary, and interaction history, and produces an updated task progress summary, an observation summary, and guidance for subsequent steps.

The *NotesvSummarizer* updates the agent's notes and their summary upon encountering a new web page. Given the current observation, prior notes, task progress summary, and action history, it extracts new task-relevant notes and generates a refined summary. These summaries help the agent monitor task progress and facilitates planning.

4.3 Planning and Execution Algorithm

Equipped with the AND/OR tree structure and the core operations described above, we now describe our modified greedy, iterative depth-first search (DFS) strategy for traversing the AND/OR planning tree. The agent begins by pushing the root node which represents the given task, onto the DFS stack in the *ENTERING* state. As in classical iterative DFS, the algorithm then iteratively pops the top node from the stack and processes it according to its current state.

When processing a node in the *ENTERING* state, the agent first checks the node's status. By default, new nodes are marked as *UNVISITED*. If the node is *UNVISITED*, the agent invokes the Node Expansion module to determine its type and expand it. For AND nodes, the module generates an ordered list of children according to task constraints. For OR nodes, it ranks alternatives by their estimated likelihood of success, as determined by LLM scoring or a value function. For ACTION nodes, it outputs the precise browser command to execute. After expansion, or if the node was previously expanded, the agent marks the node as *VISITED* and re-adds it to the stack in the *EXITING* state.

Next, the agent proceeds according to node type. For AND nodes, it pushes each child onto the stack in the *ENTERING* state for further processing. For OR nodes, it pushes only the most promising unprocessed child in the *ENTERING* state. If the node has no children, no further action is taken until its status is evaluated in the *EXITING* state. For ACTION nodes, the agent immediately executes the specified browser command. If the command is successful, the agent marks the node as *SUCCESS*, updates its internal context via the Observation Summarization module, and invokes the Note Extraction and Summarization module to refine the task summary. The Global Tree Update module prunes irrelevant branches and refines node descriptions as needed, propagating these changes upward to keep the stack and tree consistent. If the ACTION node fails, the agent marks it as *FAILED*.

When processing a node in the *EXITING* state, the agent evaluates completion criteria based on the node type. For AND nodes, the agent verifies that all valid, that is, unpruned or undeleted, child nodes have succeeded and invokes the Node Completion module to ensure that the overall objective of the AND node is met. Although this check is theoretically applicable to all AND nodes, we found it to be overly strict in practice. Therefore, to avoid unnecessary retries and failures, we apply this strict completion check only at the root node in our implementation. If completion is verified, the agent marks the node as *SUCCESS*, otherwise, as *FAILED*. For OR nodes, the agent checks if at least one child has succeeded, marking the node as *SUCCESS* if so, and as *FAILED* otherwise. For ACTION nodes, the agent sets the status to *SUCCESS* or *FAILED* depending on the outcome of the browser operation. Once a node is marked as *SUCCESS*, it is not processed again.

If the agent processes a node in the *FAILED* state, it attempts to repair the node or marks it as pruned. For ACTION nodes, it prunes the node and updates the stack. If a failed ACTION node belongs to an AND node, the agent deletes all remaining unexecuted siblings, since the conjunctive requirement cannot be satisfied. For AND nodes, if none of the children have achieved the node's objective, the agent checks the node's revision count. If the revision limit is not reached, the Node Repair module attempts to generate new subgoals or modify the set of children. If repair fails or the revision limit is reached, the agent prunes the AND node and all its descendants, propagating failure upward and updating the stack. For OR nodes in the *FAILED* state, the agent first checks for any remaining unprocessed children. If such children exist, it selects and pushes the most promising one onto the stack in the *ENTERING* state. If all alternatives are exhausted, it attempts node repair if within the revision limit, otherwise, it prunes the node. As with AND nodes, any pruning or failure of an OR node propagates upward, ensuring the global plan remains consistent and that failures at lower levels trigger repair or pruning at higher levels.

This DFS-based algorithm allows the agent to incrementally build and adapt its hierarchical plan, efficiently respond to new observations and failures, while executing the task.

4.4 Structured Memory

While prior work such as *AgentOccam* allows agents to take notes and track task progress, it does not reliably retain candidate entities identified during exploration. As a result, agents may lose valuable information discovered earlier in the search process, which can lead to revisiting previously rejected options or failing to explore alternative candidates when initial attempts are unsuccessful.

To address this limitation, we introduce a *Structured Memory* module specifically designed for information-seeking scenarios, such as recommendation tasks or research-based tasks, where the agent must retrieve information about items that satisfy user-specified constraints. In this context, candidate entities refer to items encountered by the agent that may potentially fulfill these constraints. The structured memory module extracts relevant constraints from the task description and organizes them in a dynamic table (See 3), with each row representing a candidate entity and columns capturing constraints, attributes, or task-specific notes. The schema remains flexible, allowing new columns to be added as additional constraints or features are encountered.

As the agent processes new or updated web pages, it prompts the language model to *ADD*, *UPDATE*, or *DELETE* candidate entries based on the latest information. During decision-making such as in the *Node Expansion* or *Node Repair* modules, the agent retrieves the top-K candidate entities that satisfy the most number of user constraints to guide its subsequent actions. This memory mechanism allows the agent to retain partially complete candidates, minimize redundant exploration, and make more informed decisions throughout the planning and execution process.

5 Experiments

In this section, we evaluate our proposed *Structured Agent* against two baseline agents on two web-browsing task benchmarks.

We conduct experiments on two datasets: (1) a *Custom Complex Shopping* dataset and (2) a subset of *WebVoyager* [He et al., 2024]. The *Complex Shopping* dataset contains 60 long-horizon Amazon shopping tasks. Each task is specified through a natural language query that encodes detailed constraints for one or more products, and the agent must generate recommendations that satisfy these constraints. Task completions typically require 10–30 interaction steps, reflecting the complexity and compositional nature of the user goals. The second dataset is a curated subset of the *WebVoyager* benchmark [He et al., 2024], comprising 129 real-world tasks that involve navigation across diverse live websites, including BBC News, ArXiv, Amazon, Coursera, GitHub, and HuggingFace. We report results separately on the Amazon subset of WebVoyager and on the remaining non-shopping tasks. Tasks requiring login credentials or raising security concerns are excluded, as the agent cannot reliably attempt them. All experiments are conducted using two backbone language models: *Claude 3.5 v2* and *Claude 3.7*. These serve as the underlying reasoning models powering the Structured Agent across all evaluations.

Metrics. Following prior work [He et al., 2024, Erdogan et al., 2025], we adopt *task completion rate* as the primary evaluation metric. This metric is assessed using the *LLM-as-a-Judge* framework, which evaluates agent trajectories based on summaries of observations, actions, notes, and final responses. The judge is instructed to first decompose each task into item-level and task-level constraints, and then assign one point per constraint satisfied, provided the evidence of satisfaction is grounded in the agent's observations, actions, and notes. Due to cost constraints, we primarily use GPT 4.1-mini as the judge. To ensure robustness, we additionally provide human evaluations and GPT 4.1 evaluations on the Amazon Shopping tasks. The full evaluation prompt provided to the LLM judge is included in Appendix B.1.

Baselines. We compare the Structured Agent against two strong baselines. The first is AgentOccam [Yang et al., 2025], a planning agent that incrementally constructs and prunes action trees. The second is Claude + Action History, which augments the Claude agent's input with the full action history. To isolate the contributions of Structured Memory and the AND/OR tree, we also evaluate two ablations. The *AND/OR Agent* removes Structured Memory, while the *Structured Memory Agent* corresponds to AgentOccam augmented with Structured Memory. All agents use Claude 3.5 or 3.7 as their backbone model.

6 Results

Table 2 compares the success rates of our proposed *Structured Agent* and its variants against *AgentOccam*, *Vanilla-Claude*, and *Claude-Action Agent*, all using Claude 3.5v2 as the backbone model. The results show that according the the GPT-4.1-mini judge model, the AND/OR Agent outperforms AgentOccam by *6.7%* on WebVoyager Amazon tasks when Claude 3.5 is used as the backbone model.

Agents	*Task Complexity* →		
	WebVoyager (Other Sites) (%)	**WebVoyager (Amazon)** (%)	**Complex Shopping** (%)
Claude-Action	78.8	73.3	27.8
AgentOccam	89.1	75	41.1
AND/OR Agent (Ours)	86.5	80	**48.3**
Structured Agent (Ours)	79	81.7	46.1
Structured Memory Agent (Ours)	**90.6**	**87.5**	**48.3**

Table 1: Success rates of agents across datasets ordered by increasing task complexity from left to right. Light background shading is applied to header cells using custom pastel colors to reflect task domains. All agents use Claude 3.5v2 as the backbone model.

Agents	Complex Shopping Tasks		Human Evaluation (%)
	GPT 4.1 mini (%)	**GPT 4.1** (%)	
Set 1: Claude 3.5			
Claude-Action	27.8	30	40
AgentOccam	41.1	41.7	56.67
Structured Agent (Ours)	46.1	43.3	65
Structured Memory Agent (Ours)	**48.3**	**46.7**	56.67
Set 2: Claude 3.7			
Claude-Action	35	31.7	-
AgentOccam	48.9	50	–
Structured Agent (Ours)	**56.5**	**55.9**	-

Table 2: Comparison of agent performance on complex shopping tasks using different evaluators.

Similarly, the Structured Agent surpasses AgentOccam by *7%* on Complex Shopping tasks. These improvements highlight the effectiveness of structured planning in handling complex environments. We also observe that StructuredAgent achieves a 5% improvement over AgentOccam when Claude 3.7 is used as the backbone model.

We further observe that incorporating both *Structured Memory* and the *AND/OR Tree* benefits performance on complex tasks, though it may lead to some degradation on easier tasks. Nonetheless, agents equipped with either Structured Memory or the AND/OR Tree consistently outperform both AgentOccam and the Claude-Action agent on complex tasks, underscoring the importance of these components in solving more challenging scenarios. On the other hand, *Claude-Action* performs the worst across most tasks. Despite Claude-Action having access to the full action history at every step, its poor performance suggests that *planning remains crucial*, even for powerful models with access to extensive contextual information.

We also observe some discrepancies between human judgment and GPT-4.1 evaluations, although the overall performance trends are similar. We hypothesize that these differences arise from the large number of constraints in complex tasks, which make agent performance more challenging to evaluate consistently. This suggests the need for further experimentation.

7 Conclusion

In summary, our framework leverages dynamic AND/OR tree-based planning and structured memory to address key limitations of current agentic LLMs in complex web-browsing tasks. Our framework enables hierarchical task decomposition, and improved constraint satisfaction, thereby enhancing the effectiveness of web-browsing agents. Our empirical results on Custom Complex shopping dataset and WebVoyager demonstrate the effectiveness of our proposed approach.

References

L. E. Erdogan, N. Lee, S. Kim, S. Moon, H. Furuta, G. Anumanchipalli, K. Keutzer, and A. Gholami. Plan-and-act: Improving planning of agents for long-horizon tasks, 2025. URL https://arxiv.org/abs/2503.09572.

H. He, W. Yao, K. Ma, W. Yu, Y. Dai, H. Zhang, Z. Lan, and D. Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.371. URL https://aclanthology.org/2024.acl-long.371/.

N. Kandpal and C. Raffel. Position: The most expensive part of an LLM *should* be its training data. In *Forty-second International Conference on Machine Learning Position Paper Track*, 2025. URL https://openreview.net/forum?id=L6RpQ1h4Nx.

J. Y. Koh, R. Lo, L. Jang, V. Duvvur, M. Lim, P.-Y. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried. VisualWebArena: Evaluating multimodal agents on realistic visual web tasks. In L.-W. Ku, A. Martins, and V. Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, Bangkok, Thailand, Aug. 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

J. Y. Koh, S. M. McAleer, D. Fried, and R. Salakhutdinov. Tree search for language model agents, 2025. URL https://openreview.net/forum?id=kpL66Mvd2a.

S. Murty, H. Zhu, D. Bahdanau, and C. D. Manning. Nnetnav: Unsupervised learning of browser agents through environment interaction in the wild, 2025. URL https://arxiv.org/abs/2410.02907.

P. Putta, E. Mills, N. Garg, S. R. Motwani, E. S. Markowitz, J. Kiseleva, C. Finn, D. Garg, and R. Rafailov. Agent q: Advanced reasoning and learning for autonomous AI agents, 2025. URL https://openreview.net/forum?id=LuytzzohTa.

Z. Qi, X. Liu, I. L. Iong, H. Lai, X. Sun, J. Sun, X. Yang, Y. Yang, S. Yao, W. Xu, J. Tang, and Y. Dong. WebRL: Training LLM web agents via self-evolving online curriculum reinforcement learning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=oVKEAFjEqv.

R. Rafailov, A. Sharma, E. Mitchell, S. Ermon, C. D. Manning, and C. Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL https://arxiv.org/abs/2305.18290.

T. Schmied, J. Bornschein, J. Grau-Moya, M. Wulfmeier, and R. Pascanu. Llms are greedy agents: Effects of rl fine-tuning on decision-making abilities, 2025. URL https://arxiv.org/abs/2504.16078.

N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

M. Simchowitz, D. Pfrommer, and A. Jadbabaie. The pitfalls of imitation learning when actions are continuous, 2025. URL https://arxiv.org/abs/2503.09722.

P. Sodhi, S. Branavan, Y. Artzi, and R. McDonald. Step: Stacked LLM policies for web actions. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=5fg0VtRxgi.

H. Wang, S. Hao, H. Dong, S. Zhang, Y. Bao, Z. Yang, and Y. Wu. Offline reinforcement learning for llm multi-step reasoning, 2024. URL https://arxiv.org/abs/2412.16145.

Z. Wang, J. Mao, D. Fried, and G. Neubig. Agent workflow memory, 2025. URL https://openreview.net/forum?id=PfYg3eRrNi.

K. Yang, Y. Liu, S. Chaudhary, R. Fakoor, P. Chaudhari, G. Karypis, and H. Rangwala. Agentoccam: A simple yet strong baseline for LLM-based web agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=oWdzUpOlkX.

M. Zare, P. M. Kebria, A. Khosravi, and S. Nahavandi. A survey of imitation learning: Algorithms, recent developments, and challenges, 2023. URL https://arxiv.org/abs/2309.02473.

Y. Zhang, Z. Ma, Y. Ma, Z. Han, Y. Wu, and V. Tresp. Webpilot: A versatile and autonomous multi-agent system for web task execution with strategic exploration, 2024. URL https://arxiv.org/abs/2408.15978.

A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024. URL https://arxiv.org/abs/2310.04406.

A Algorithm Details

Algorithm 1: and_or_tree_traversal(root)

```
stack.append((root, NodeState.ENTERING))
counter ← 0
while stack not empty do
    node, state ← stack.pop()
    if node.status == NodeStatus.PRUNED then
        self.backtrack_failure(node)
        continue
    end
    else if node.status == NodeStatus.DELETED then
        continue
    end
    if state == NodeState.ENTERING then
        stack, node ← process_node_entering(node, stack)
    end
    else if state == NodeState.EXITING then
        stack, node ← process_node_exiting(node, stack)
    end
    else if state == NodeState.FAILED then
        stack, node ← process_node_failed(node, stack)
    end
    if counter ≥ BUDGET then
        break
    end
end
```

Algorithm 2: process_node_entering(node, stack)

```
if node.parent and node.parent.type == NodeType.OR then
    perform_context_rollback(node.parent)
end
set_context(node)
node.execution_count ← node.execution_count + 1
if node.type == NodeType.UNKNOWN then
    node ← populate_node_type(node)
    stack.append((node, NodeState.EXITING))
end
if node.type == NodeType.ACTION then
    stack.append((node, NodeState.EXITING))
    status ← perform_action(node)
    if status == SUCCESS then
        perform_global_tree_update()
        update_notes_and_observations()
        node.status ← NodeStatus.SUCCESS
    end
    else
        node.status ← NodeStatus.FAIL
    end
end
if node.type == NodeType.AND then
    if is_successful(node) then
        continue
    end
444 else if has_atleast_one_valid_child(node) then
    foreach child Input: r
    everse(node.children) do
        if child.status not
        in CLOSED_STATUSES then
            stack.append((child, NodeState.ENTERING))
        end
    end
    end
    else
        node.status ← NodeStatus.FAIL
    end
end
if node.type == NodeType.OR then
    success ← is_successful(node)
    if success then
        continue
    end
    else if is_valid(node) then
        child ← find_next_promising(node.children)
        stack.append((child, NodeState.ENTERING))
    end
    else
        node.status ← NodeStatus.FAIL
    end
end
return stack, node
```

Algorithm 3: process_node_exiting(node, stack)

```
if node.type == NodeType.ACTION then
    if node.status Input: F
    AILED_OR_PRUNED then
        stack.append((node, NodeState.FAILED))
    end
end
else
    node.status ← NodeStatus.SUCCESS
end
if node.type == NodeType.AND then
    success ← is_successful_and(node)
    if success then
        is_complete ← check_for_completion_and(node)
        if is_complete then
            node.status ← NodeStatus.SUCCESS
            continue
        end
        else
            node.status ← NodeStatus.FAILED
            stack.append((node, NodeState.FAILED))
        end
    end
    else
        node.status ← NodeStatus.FAILED
        stack.append((node, NodeState.FAILED))
    end
end
else if node.type == NodeType.OR then
    success ← is_successful_or(node)
    if success then
        node.status ← NodeStatus.SUCCESS
    end
    else
        node.status ← NodeStatus.FAIL
        stack.append((node, NodeState.FAILED))
    end
end
return stack, node
```

Algorithm 4: process_node_failed(node, stack)

```
if node.type == NodeType.ACTION then
    node.status ← NodeStatus.PRUNED
    self.propagate_failure_in_tree(node)
end
else if node.type == NodeType.AND then
    if has_atleast_one_success then
        is_complete ← check_for_completion_and(node)
        if is_complete then
            node.status ← NodeStatus.SUCCESS
            continue
        end
    end
    is_valid ← is_valid_and(node)
    if not is_valid then
        if node.revision_count < MAX_REVISION_COUNT then
            revised ← revise_and_node(node)
            stack ← synchronize_stack()
            if node.status == NodeStatus.SUCCESS then
                continue
            end
            else if revised or is_valid then
                node.status ← NodeStatus.VISITED
                self.stack.append((node, NodeState.ENTERING))
            end
            else
                prune_node(node)
                stack ← synchronize_stack(stack)
                self.propagate_failure_in_tree(node)
            end
        end
    end
end
else if node.type == NodeType.OR then
    is_valid ← is_valid_or(node.children)
    if is_valid then
        node.status ← NodeStatus.VISITED
        self.stack.append((node, NodeState.ENTERING))
        continue
    end
    else
        if node.revision_count < MAX_REVISION_COUNT then
            revised ← revise_or_node(node)
            stack ← synchronize_stack(node)
            if revised then
                node.status ← NodeStatus.VISITED
                self.stack.append((node, NodeState.ENTERING))
            end
            else
                prune_node(node)
                stack ← synchronize_stack(node)
                self.propagate_failure_in_tree(node)
            end
        end
    end
end
return stack, node
```

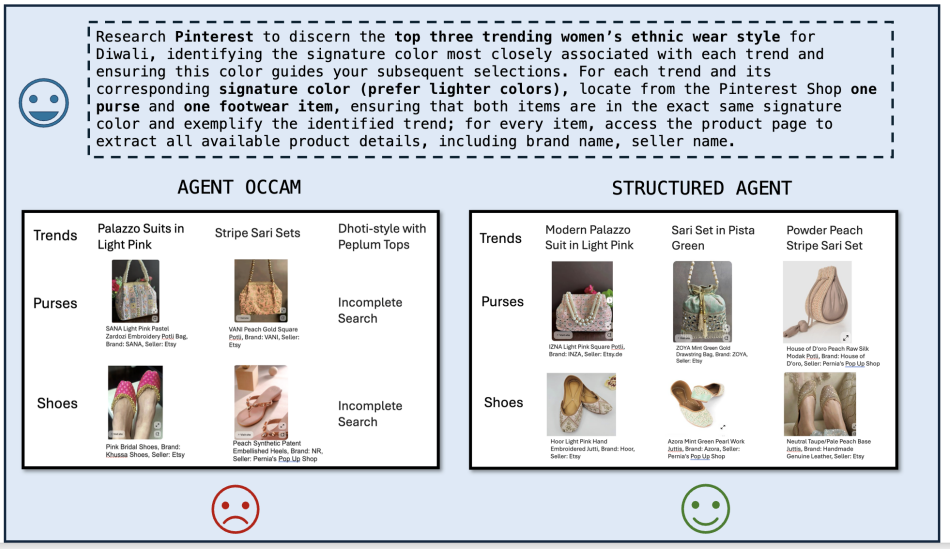

Figure 4: AgentOccam vs Structured Agent on a Pinterest task: AgentOccam terminates its search prematurely when it fails to explore the third trend. On the other hand, StructuredAgent is able to successfully retrieve items for all three trending women's ethnic wear styles.

B Planning Tree

- [0] (AND) Research Pinterest to identify the top 3 women's ethnic wear trends for Diwali. For each trend: 1) Specify the signature color currently trending. 2) Find one matching footwear style and one purse, both in the same color (same color as the trend) from the Pinterest Shop. 3) Provide the name of a potential brand or seller for each footwear and purse, and include a direct source or pin link if possible. Focus on vibrant, festive, and popular looks. Do not end your search early. Limit your review to only the first 2 scrolls of results. Do not click on external links. [success]
- [0.1] (Atomic) Search for Diwali ethnic wear trends on Pinterest (type [401] [Diwali ethnic wear trends 2024] [1]) [success]
- [0.2] (Atomic) Document 3 trending ethnic wear styles with their signature elements (note [Trending Diwali Ethnic Wear 2024: 1. Pink - Anupa Sari with embellished blouse from INDIASPOPUP.COM 2. Pista Green - Crush fabric sari with sequins embroidery border and yellow jacquard blouse from Etsy 3. Powder Peach - Stripe sari set with thread, pearl and sequin embroidery, featuring plunging neckline blouse from KYNAH]) [success]
- [0.3] (AND) Navigate to Pinterest Shop and search for footwear in Pink, Pista Green, and Powder Peach shades from verified sellers specifically for Diwali looks [success]
- [0.3.1] (Atomic) Navigate to Pinterest Shop section (click [208]) [success]
- [0.3.2] (Atomic) type [401] [Pink ethnic juttis Diwali footwear] [1] (type [401] [Pink ethnic juttis Diwali footwear] [1]) [success]
- [0.3.3] (Atomic) Return to previous page after completing pink footwear search (go_back) [success]
- [0.3.4] (Atomic) type [401] [Pista Green ethnic juttis Diwali] [1] (type [401] [Pista Green ethnic juttis Diwali] [1]) [success]
- [0.3.5] (Atomic) Return to previous page to continue accessory search (go_back) [success]
- [0.3.6] (Atomic) type [401] [Powder Peach ethnic juttis Diwali] [1] (type [401] [Powder Peach ethnic juttis Diwali] [1]) [success]
- [0.4] (OR) Search Pinterest Shop for matching ethnic clutches/purses in three colors by trying alternative search terms: 1) Search "Pink potli Diwali bag" or "Pink ethnic clutch" from verified sellers, 2) Search "Pista Green potli Diwali bag" or "Mint ethnic clutch" from verified sellers, 3) Search "Peach potli Diwali bag" or "Blush ethnic clutch" from verified sellers. Limit to first 2 scrolls of results. [success]
- [0.4.1] (AND) Search for powder peach ethnic clutches in Shop section [success]
- [0.4.1.1] (Atomic) Search for powder peach ethnic clutches in Shop section (type [401] [powder peach ethnic clutch Diwali] [1]) [success]
- [0.4.1.2] (Atomic) Execute search query for powder peach ethnic clutches (type [401] [powder peach ethnic potli clutch bag Diwali] [1]) [success]
- [0.4.1.4] (Atomic) note [Found matching powder peach potli bags from verified sellers: 1. 5 Elements Peach Brocade Pattern Potli Bag at Pernia's Pop Up Shop 2. AMYRA Peach Poly Silk Pearl Embroidered Potli Bag 3. SWABHIMANN ACCESSORIES Peach Pure Silk Potli Bag] (note [Found matching powder peach potli bags from verified sellers: 1. 5 Elements Peach Brocade Pattern Potli Bag at Pernia's Pop Up Shop 2. AMYRA Peach Poly Silk Pearl Embroidered Potli Bag 3. SWABHIMANN ACCESSORIES Peach Pure Silk Potli Bag]) [success]
- [0.4.1.5] (Atomic) Modify search query to find matching powder peach potli bags (type [401] [powder peach potli bag Diwali] [1]) [success]
- [0.4.1.6] (Atomic) click [208] to ensure Shop filter is active before searching for powder peach accessories (click [208] to ensure Shop filter is active before searching for powder peach accessories) [success]
- [0.4.1.8] (Atomic) Modify search to find powder peach potli bags for Diwali (type [401] [powder peach potli bag Diwali] [1]) [success]
- [0.5] (Atomic) Document powder peach accessory pairs with seller details (note [Powder Peach - Footwear: Coral Haze Juttis, Jutti Express formal wear, 5 Elements ethnic shoes; Purse: 5 Elements Brocade Potli with tassels, AMYRA Poly silk Pearl Embroidered Potli, Ri Ritu Kumar Net and Silk Zari Potli]) [success]
- [0.6] (AND) Document pink and pista green accessory pairs with seller details and pin links, ensuring color coordination and authenticity of sellers [success]
- [0.6.1] (AND) Search and document pink ethnic footwear with seller details [success]
- [0.6.1.1] (Atomic) Search for pink ethnic footwear for Diwali (type [401] [pink ethnic juttis Diwali] [1]) [success]
- [0.6.1.2] (Atomic) note [Found matching pink footwear options: 1. B Anu Designs - Baby Pink Punjabi Jutti with Mirror and Bead handwork 2. Miar Designs - Pink Leather Embroidered Juttis at Pernia's Pop Up Shop 3. YASSIO - Pink Silk Floral Embroidered Juttis] (note [Found matching pink footwear options: 1. B Anu Designs - Baby Pink Punjabi Jutti with Mirror and Bead handwork 2. Miar Designs - Pink Leather Embroidered Juttis at Pernia's Pop Up Shop 3. YASSIO - Pink Silk Floral Embroidered Juttis]) [success]
- [0.6.1.3] (Atomic) Return to previous page to continue accessory search (go_back) [success]
- [0.6.2] (Atomic) Document matching pink potli/clutch from search results (note [Found matching pink brocade potli bag from 5 Elements with tassels and beads embellishments, suitable for festive wear]) [success]
- [0.6.3] (Atomic) Return to previous page after powder peach potli search (go_back) [success]
- [0.6.4] (Atomic) Search for pista green ethnic footwear using specific search terms (type [401] [Pista green embroidered juttis Diwali] [1]) [success]
- [0.6.5] (Atomic) Search for matching pista green clutch/potli (type [401] [Pista green brocade potli Diwali] [1]) [success]
- [0.6.6] (Atomic) Document complete accessory details with seller information for pista green trend accessories: Footwear - House of Vian's Crepe Embroidered Juttis and 5 Elements' Leatherette Embellished Juttis; Purse - 5 Elements Green Brocade Embroidered Potli with tassels and dupion lining, and Tarini Nirula Green Brocade Stone Embellished Potli with antique gold frame (Document complete accessory details with seller information for pista green trend accessories: Footwear - House of Vian's Crepe Embroidered Juttis and 5 Elements' Leatherette Embellished Juttis; Purse - 5 Elements Green Brocade Embroidered Potli with tassels and dupion lining, and Tarini Nirula Green Brocade Stone Embellished Potli with antique gold frame) [success]
- [0.6.7] (Atomic) Search for powder peach ethnic footwear (type [401] [powder peach ethnic juttis Diwali] [1]) [success]
- [0.6.10] (AND) Search for pink ethnic footwear pins from verified sellers and save direct links [success]
- [0.6.10.1] (Atomic) View details of pink leather embroidered juttis for Pink category (click [17999]) [success]
- [0.6.10.2] (Atomic) note [Pink leather juttis by Miar Designs - Features: Leather base with coin, thread, pearl, sequins and beads embroidery, Available at Pernia's Pop Up Shop, Price: 40,00, Formal occasion wear] (note [Pink leather juttis by Miar Designs - Features: Leather base with coin, thread, pearl, sequins and beads embroidery, Available at Pernia's Pop Up Shop, Price: 40,00, Formal occasion wear]) [success]
- [0.6.10.3] (Atomic) Return to previous page to continue purse search (go_back) [success]
- [0.6.11] (Atomic) Search for matching pink potli/clutch pins (type [401] [pink ethnic potli clutch Diwali] [1]) [success]
- [0.6.14] (Atomic) Verify and document color coordination between footwear-purse pairs (note [Color coordination verified: Pink category - Pink Beaded Mirror Work Potli and Pink Brocade Potli perfectly match the pink leather juttis from Miar Designs in shade and festive style]) [success]
- [0.7] (Atomic) Document direct pin links for pink accessories from verified sellers (note [Found verified seller pin links: 1. Pink Brocade Tassels Embellished Potli by 5 Elements - Direct Pin ID 20730 2. Pink Brocade Ghungroo Embroidered Potli by 5 Elements - Direct Pin ID 20752 3. Pink Beaded Mirror Work Potli by Etsy verified seller - Direct Pin ID 21242]) [success]
- [0.8] (AND) Review and document direct pin links for all three color categories ensuring each has verified footwear and purse pairs with seller details [success]
- [0.8.1] (Atomic) Document completed findings for Pink category (note [Pink Category Complete - Footwear: Miar Designs, Purses: Pink Beaded Mirror Work Potli (Etsy), Pink Brocade Potli (5 Elements)]) [success]
- [0.8.2] (Atomic) Navigate to Pinterest Shop section for ethnic potli clutches (click [20411]) [success]
- [0.8.7] (Atomic) Search for powder peach potli options (type [401] [powder peach ethnic potli clutch Diwali] [1]) [success]
- [0.8.8] (Atomic) Search for matching Pista Green potli options with verified sellers and clear product details (type [401] [pista green ethnic potli clutch Diwali] [1]) (Search for matching Pista Green potli options with verified sellers and clear product details (type [401] [pista green ethnic potli clutch Diwali] [1])) [success]
- [0.8.9] (Atomic) Document verified seller information for Pista Green potlis (note [Verified Pista Green Potlis: 1. PLODE - Green Velvet Swarovski Triangle Potli with crystal embellishments and rhinestone strap 2. Velleso - Green Mirror Work Potli with crystal drops and silver threadwork (24x23cm) 3. The Pink Potli - Green Raw Silk Potli with bead work and tassel embellishments]) [success]
- [0.8.10] (Atomic) Search for matching Powder Peach ethnic footwear options (type [401] [powder peach ethnic footwear Diwali] [1]) [success]
- [0.8.11] (Atomic) Search for matching Pista Green ethnic footwear options with verified sellers and clear product details (type [401] [pista green ethnic footwear Diwali] [1]) [success]
- [0.8.12] (Atomic) Document pin links and details for Pista Green category footwear and purse pairs (note [Found direct pin links for Pista Green category: 1) House of Vian Green Crepe Embroidered Juttis with crystals and gota patti from Pernia's Pop Up Shop 2025, 2) 5 Elements Green Leatherette Embellished Juttis with beadwork from Pernia's Pop Up Shop 2025]) [success]

Table 3: Planning tree that was dynamically constructed for for identifying Diwali ethnic wear trends on a Pinterest task

B.1 Prompts

You are a web-browsing assistant designed to extract structured constraint data from user queries. Given a natural language query, your task is to identify and return:
1. A list of **distinct items** the user wants to retrieve (if any). 2. For each item, a list of **item-level constraints** that are explicitly mentioned in the query. 3. A separate list of **task-level constraints** that apply to the overall task.
Definitions:
- **Item**: A specific product, object, or entity that the user wants to find or retrieve information about. - **Item-level constraint**: Any attribute or specification that is directly tied to an individual item. This may include (but is not limited to): brand, model, price, rating, features, color, size, dimensions, delivery time, material, and any other **explicitly** stated property of the item. - **Task-level constraint**: A condition that affects the overall search or task execution rather than a specific item.
Instructions:
- You must extract **only those task-level and item-level constraints that are explicitly stated** in the query. - You must **not assume, infer, or interpret** any implicit constraints, even if they seem logically implied or contextually likely. - Do **not include any reasoning, commentary, or explanation** in your output. - Each item type must appear only once in the 'ITEM_CONSTRAINTS' list. - If the query contains no valid constraints or items, return empty lists using the required format.
Output Format:
The result must be returned strictly in the following JSON format:
{ "TASK_CONSTRAINTS": ["explicit_task_constraint1", "explicit_task_constraint2", "..."], "ITEM_CONSTRAINTS": [{ "item_type": ["explicit_item_constraint1", "explicit_item_constraint2", "..."] }, ...] }
EXAMPLE:
QUERY: Find a black laptop bag under $40 and a wireless mouse under 25. Both should be delivered within 2 days. URL: www.amazon.com
OUTPUT:
{ "TASK_CONSTRAINTS": [ ], "ITEM_CONSTRAINTS": [{ "laptop bag": ["Color: black", "Price: under $40", "Delivery time: within 2 days"] }, { "wireless mouse": ["Price: under $25", "Type: wireless", "Delivery time: within 2 days"] }] }
QUERY: {task_objective}

Table 4: Prompt for extracting task constraints for task completion evaluation.

We propose the *Subplan Evaluator* as an auxiliary module to identify and discard low-quality subplans generated by the agent. Recall that the planner uses an LLM to expand AND and OR nodes. For an AND node, its children represent a subplan-i.e., a sequence of high-level steps intended to fulfill the node's objective.

In our experiments, we observed that LLM-generated subplans are occasionally incomplete, vague, or contain subgoals that are not actionable. For instance, the LLM may incorrectly specify an actionable subgoal as a `note` atomic action. This results in subplans that fail to progress toward the objective within the web-browser environment, as the agent merely records information rather than taking action.

To mitigate the impact of such poor subplans, we introduce a reward model that evaluates subplans based on criteria such as completeness, logical ordering, redundancy, and correctness. During node expansion, the planner can sample multiple candidate subplans and use this reward model to select the most promising one.

We describe the training procedure for this reward model using a corpus of successful trajectories provided by Murty et al. [2025].

C Related Works..continued

Fine-Tuned LLMs as Web Agents: Several studies [Erdogan et al., 2025, Qi et al., 2025, Murty et al., 2025] have explored the use of open-source language models for web-based tasks, however, these models frequently exhibit high failure rates due to their limited capacity and insufficient domain-specific knowledge, stemming from their relatively small sizes. To address this, later works Erdogan et al. [2025], Murty et al. [2025] proposed fine-tuning open-source models on expert trajectories generated either by humans or closed-source LLMs. These studies show that such fine-tuning these models on expert trajectories significantly improves performance, particularly when test tasks are sampled from the same distribution as the training data. These methods commonly employ either imitation learning [Zare et al., 2023], where agents are trained on offline expert trajectories, or reinforcement learning (RL), where agents learn through online interactions. However, agents trained via imitation learning merely maximize the likelihood of expert actions, making them brittle and prone to failure on complex or out-of-distribution tasks [Simchowitz et al., 2025]. Their effectiveness heavily depends on the quality and diversity of the training trajectories, which are often expensive to collect, especially when using proprietary LLMs. On the other hand, training web agents with online RL is computationally intensive [Kandpal and Raffel, 2025, Wang et al., 2024]. Moreover, these agents typically receive only sparse binary feedback based on task or episode completion, which introduces a credit assignment problem in partially correct trajectories. For example, the RL algorithm may penalize correct intermediate actions if the final outcome is a task failure. As a result,

You are an evaluator assessing the performance of a web-browsing agent on a specific task.
You will be provided with the following components:
- Task Description A natural-language instruction outlining the goal the agent is expected to accomplish. This may involve retrieving, summarizing, comparing, or verifying information from the web.
- Constraints A list of specific item-level and task-level requirements that must be satisfied for the task to be considered successful.
- Trajectory Summary A step-by-step log of the agent's interaction with the environment. Each time step includes: - Observation: Summary of the web-page the agent perceives at that moment. - Action: The behavior or command the agent executes (e.g., clicking, searching). - Interaction History Summary: Summary of the agent's progress (if available)
- The agent's final response to the task
Evaluation Guidelines
- **Evaluate each item-level constraint once per item type**. Do not repeat the constraint evaluation for each instance of the same item type. Instead, assess whether the constraint is satisfied **across all relevant items of that type**, and justify the evaluation with evidence from the observations, notes, or interaction history. **For example:** If the task requires finding 3 laptops under $1000, and the agent recommends 3 laptops, evaluate the constraint "Price : under 1000" only once. Mark it as **satisfied only if all 3 laptops** meet the price constraint.
- Only use information that is **explicitly visible** in the agent's observations, actions, notes, and final response. Do not rely on external knowledge, domain expertise, or assumptions made by the agent.
- Do **not assume that a constraint is satisfied** based on the agent's search terms, reasoning, or keyword usage (e.g., searching for "size 6"). A search query does **not guarantee** that the results meet the constraint. A constraint is only considered satisfied if the specific value (e.g., "Size: 6") is explicitly shown in the observations, actions or agent's notes.
- If the task requires finding the cheapest, most rated, or most reviewed item, only verify that appropriate filters or sorting options were applied - but **only if those options are actually available on the website**.
- Evaluate only based on the **final response** of the agent. Intermediate steps or partially explored options should not be considered unless they directly contribute to the final result.
- Do **not award partial credit** for incomplete constraints. A constraint is either **fully satisfied** or **not satisfied**. All requested attribute values (e.g., price, brand, size) are treated as **separate constraints**.
- Do **not evaluate** the agent based on constraints that are **not explicitly stated** in the task description. For example, if the task does not mention "size," "material," or "capacity," do not treat them as constraints — even if the agent includes or discusses them during the task.
- If there is no final response or the task is incomplete, assign a score of 0.
- You do not need to consider constraints that require the agent to find items with a given delivery period as the agent does not have access to the current date and time.
Scoring Instructions
- Assign **1 point** for each item-level or task-level constraint that is fully and explicitly grounded in the observations, notes and actions. **Confirm that the agent is not hallucinating.** **Note:** If the recommended item costs **less than** the specified price, it still satisfies the price constraint. Evidence for constraints satisfied must be grounded only in observations, notes and actions. **Do not** ground evidence in the final response of the agent.
- Assign **0 points** for constraints that are: - Not satisfied or Partially Satisfied - Assumed or inferred without explicit evidence in observations, notes, or final response
- Assign **1 point** if the task was completed successfully and the final response is likely **not hallucinated**. A task is considered **completed successfully** if the agent provides a clear and specific final response that directly fulfills the task objective, and is **explicitly supported** by observed content or notes.
- Normalize the score: **Score = (Number of Satisfied Constraints + 1 point if task completed) / (Total Number of Constraints + 1)** This yields a final score between **0.0 and 1.0**.
Final Verdict
- **SUCCESS** – Only if **all** explicitly stated item and task-level constraints are fully satisfied (score = 1.0) - **FAILURE** – If **any** constraint is not satisfied, or if the task is incomplete (score < 1.0)
Your response must strictly follow this format:
1. **Task Overview** - **Task Description**: (Copy the task as provided.)
- **Constraints**: (List the item-level and task-level constraints exactly as stated in the task description. **Do not infer or add any additional constraints.**)
2. **Evaluation of Each Constraint** Evaluate each constraint using one of the following labels: - Satisfied - Partially Satisfied - Not Satisfied
Constraint 1: Evidence: Evaluation: Reason:
Constraint 2: Evidence: Evaluation: Reason:
(Continue for all constraints.) **Note**: Evidence must be grounded in observations, notes and actions only. Evidence cannot be grounded in the final response of the agent.
3. **Agent Behavior Assessment** - **Action Effectiveness**: Was the agent's sequence of actions appropriate and goal-directed?
- **Interpretation Accuracy**: Did the agent correctly interpret the content it observed?
- **Handling of Multi-step Tasks (if applicable)**: Did the agent address all required components of the task?
4. **Final Scoring** - Total Satisfied Constraints: X - Total Constraints: Y - Score (X / Y): Z - Final Verdict: SUCCESS / FAILURE
5. **Summary Comments (Optional)** (Provide any additional insights or suggestions about the agent's behavior or reasoning.)
OBJECTIVE: {objective}
CONSTRAINTS: {constraints}
FINAL RESPONSE: {notes}
TRAJECTORY {trajectory}

Table 5: Prompt for evaluating completion of task based on observation summaries, actions, notes and final response of the agent.

You are an efficient AND/OR Tree Constructing Agent specialized in web-browsing tasks. You solve complex problems using AND/OR planning trees. You dynamically construct AND/OR planning trees from observations of the webpage's HTML DOM structure for efficient and robust task execution.
You are provided:
- root-level task description
- task constraints at item level
- HTML DOM structure as the observation
- node_id and desription of the node to analyze
- summary of current task progress
- summary of notes taken so far
- Information about node's siblings and node's parent's siblings.
Node definitions:
Node Types <AND/ OR / Atomic>
- AND Node: Represents an ordered list of logical subgoals required to achieve the node's objective.
- OR Node: Represents alternative sub-strategies (which can be other AND/OR nodes)
- Atomic Action: Single executable action strictly matching navigation_specifications described below.
Node status indicators:
- Unvisited nodes marked UNVISITED
- Pruned nodes marked PRUNED
- Completed nodes marked SUCCESS
- Temporarily failed nodes marked FAIL
NAVIGATION_SPECIFICATIONS: <LIST OF BROWSER ACTIONS>
Your Task
For the given node:
1. First, briefly analyze the node type: Based on the available information and the node description, determine whether the node is an AND node, an OR node, or an Atomic node. Justify your reasoning before taking any action.
2. Then, choose ONE of the following options for the node:
A. MARK node as Atomic if: - the goal can be achieved using a single atomic action that strictly matches the NAVIGATION_SPECIFICATION. OR - the goal does not require any browser navigation actions like 'type', 'click', and 'go_back' and is information retrieved from thr webpage that can be noted down using the 'note' action.
B. EXPAND the node if: - the goal is not atomic and requires executing a sequence of atomic actions from NAVIGATION_SPECIFICATIONS. Clearly state the node type (AND/OR): - Do not consider sub-goals and sub-strategies that cannot be executed on the webpage using one action or a sequence of actions from NAVIGATION_SPECIFICATIONS. - For AND nodes, provide ordered list of logical subgoals that are necessary for satisfying the node's objective. - For OR nodes, provide observable list of alternative sub-strategies only. - For OR nodes, order alternatives by likelihood of success, assigning scores within the range (0,1). - Do not add speculative or redundant subgoals.
VERY IMPORTANT RULES: - **Focus on expanding nodes in a way that will result in completing the task faster with high probability (Don't take very risky actions)**.
- Make sure that the temporal order of the children of an AND node is correct, detailed, and efficient. - Make sure to not split notes into multiple atomic actions.
- Strictly do not use 'note' for noting future actions or subgoals. For example, this is not a note -> '[Need to examine each search result for: 1. Publication date 2024 2. Review count 20+ 3. Confirm it's a Japan travel guide book]'
- Use 'go_back' action to return to previous page especially when you have consecutive atomic actions like 'click', 'type', 'goto'.
- Atomic actions like 'type', 'click' must be accompanied with valid IDs from the HTML DOM observation.
- Use atomic action 'note' strictly to note information that is necessary for completing the task or for actions that cannot be executed on the webpage. - Expand only non-atomic steps. Do not decompose goal when it is directly achievable using an action from NAVIGATION_SPECIFICATIONS.
- Create OR nodes to consider different alternative strategies in cases where failure is probable.
- Structure plans as a knowledgeable efficient human would.
Output Format:
Begin by briefly describing the key elements from the input (overall task description, task progress, task feedback, task notes, node description, observation). Then analyze the node within the given context and clearly identifying its type.
Do not change the node's intended objective or assume a different node objective. Your highest priority is satisfying the node's objective.
You can borrow notes from the notes summary if it helps in achieving the node's objective. Use the local tree information to ensure that you are not repeating any subgoals that have already been considered.
If the node is an Atomic Action, determine and briefly analyze the 3 best possible actions it could represent. For each, explain why it fits the context. Then, select the single best action among them that is not very risky and will contribute to faster completion of the task and justify your choice.
Conclude your response using ONE of the following formats:
FORMAT 1 (Enclose all values in «» on a single line):
Node ID: «node_id»
Node Description: «Describe what does the atomic action do»
Node Type: «Atomic»
Expansion: «Exact Atomic action from NAVIGATION_SPECIFICATIONS»
Reasoning: «Brief justification explaining why the node is classified as Atomic» OR
FORMAT 2 (Enclose all values in «» on a single line, colon (;) separated):
Node ID: «node_id»
Node Description: «node_description»
Node Type: «AND / OR»
Expansion: «1. First subgoal; 2. Second subgoal; 3. Additional subgoals» OR «1. Alternative strategy; 2. Alternative strategy; Additional alternatives as necessary;...»
Reasoning: «Brief justification explaining why the node is classified as AND, OR, or Atomic, based on current observations.»
Example 1: Node ID: «2.1»
Node Description: «Find recipe with visible ratings and review counts»
Node Type: «Atomic»
Expansion: «click [123]»
Example 2:
Node ID: «2»
Node Description: «Filter results by rating and number of reviews»
Node Type: «AND»
Expansion: «1. click [Recipe with 69 ratings to check star rating]; 2. go_back; 3. click [Check page 2 if first recipe doesn't meet criteria]; 4. go_back; 5. click [Check page 3 if needed]»
Example 3:
Node ID: «0.5.1»
Node Description: «Filter recipes by rating and review count»
Node Type: «OR»
Expansion: «1. Manually scan recipe listings for review counts and ratings (score: 0.9), 2. Click through to individual recipes to check ratings and reviews (score: 0.8)»
Do not output anything outside of the specified format. Atomic actions must strictly match format specified in NAVIGATION_SPECICATIONS.
NOTE: Atomic actions 'click' and 'type' must be accompanied with valid IDs from the HTML DOM observation.
ROOT-LEVEL TASK DESCRIPTION:
{task_description}
ITEM LEVEL CONSTRAINTS:
{item_constraints} {additional_context}
OBSERVATION: {observation}
NODE_ID : Description: node_id : {node_description}
LOCAL TREE INFORMATION: {local_tree_info}

Table 6: Prompt for node expansion.

You are an AND/OR Tree Completeness Evaluator Agent specialized in web-browsing tasks. Your role is to assess whether a given AND node's objective has been successfully executed by the child nodes.
Node definitions:
Node Types <AND/ OR / Atomic>
- AND Node: Represents an ordered list of logical subgoals required to achieve the node's objective.
- OR Node: Represents alternative sub-strategies (which can be other AND/OR nodes)
- Atomic Action: Single executable action strictly matching navigation_specifications described below.
Node status indicators:
- Unvisited nodes marked UNVISITED
- Pruned nodes marked PRUNED
- Completed nodes marked SUCCESS
- Temporarily failed nodes marked FAIL
NAVIGATION_SPECIFICATIONS:
<LIST OF BROWSER ACTIONS>
You are provided:
- The root-level task description
- Item level constraints if any
- node_id and description of the node to analyze
- Task progress summary
- Notes summary: notes taken by the agent so far - A list of its children: each with an ID, description, and status (SUCCESS, VISITED, UNVISITED, or PRUNED)
- Task feedback
Your task is to determine whether a given AND node is complete.
Follow the steps below to make this determination:
1. Examine the current children of the AND node and their statuses.
- Identify each child subgoal or atomic action.
- Check whether each is marked as SUCCESS, FAILURE, or INCOMPLETE.
2. Review the task progress, task feedback, and notes summary.
- Look for any additional information that confirms or contradicts the completion of the node's objective.
3. Determine whether all required subgoals or actions are present.
- Ensure no required subgoals are missing.
- Confirm that the subgoals appear in the correct order, as expected based on the task tree and the overall task objective.
4. Verify that each required subgoal or atomic action is marked as SUCCESS.
- Do not assume completion unless it is explicitly indicated.
- Partial or INCOMPLETE statuses are not sufficient.
5. Conclude that the AND node is COMPLETE only if: - All required subgoals or atomic actions are present, in the correct order, and marked SUCCESS; OR - The successful child nodes, task progress summary, and task feedback clearly indicate that the node's objective has been fully achieved.
- **You are only required to satisfy explicitly stated constraints. Do not assume or enforce any implicit constraints.**
6. Mark the AND node as INCOMPLETE if:
- The node's objective is not fully achieved, or
- Any required subgoal, step, or relevant information is missing or not marked SUCCESS.
Special Case - Root Node Evaluation:
If the node being evaluated is the root node, you must additionally verify whether the overall task objective is fully satisfied.
- If the task requires recommending or evaluating items, then:
- You must iterate over each item that is being considered or presented as a recommendation.
- For each item, check whether it satisfies all task constraints explicitly provided in the task description or item-level constraints.
- Note that constraints that are not explicitly mentioned in the task need not be satisfied.
- The node is incomplete if any recommended item fails to meet required constraints or if evaluation is missing or partial.
- The node is incomplete if the task is incomplete and some information explicitly required by the task is missing.
- Do not assume any additional implicit constraints than the ones explicitly specified.
Begin by analyzing all the information provided (task progress summary, notes summary, node description, children description).
Provide a brief explanation for your decision, citing any missing, out-of-order, or incomplete subgoals if relevant.
End your response in strictly one of the following formats, using a single line (reasoning and node id must be enclosed in «»):
If node's objective is achieved:
COMPLETE «node_id»
Reasoning: «Reason why node's objective is achieved by the agent. Cite references from task progress summary, notes summary and successful children's description»
OR
If node's objective is not achieved:
INCOMPLETE «node_id»
Reasoning: «Reason why node's objective is not achieved by the agent, Cite references from task progress summary, notes summary and successful children's description»
Example 1:
COMPLETE «0.1»
Reasoning: «All required subgoals for finding a compact digital camera with specified requirements are present, in logical order, and marked as SUCCESS.»
Example 2:
INCOMPLETE «0.2»
Reasoning: «The subgoal to compare prices is marked as PRUNED, and the subgoal to check reviews is missing. The AND node's objective is not fully achieved.»
You must strictly follow the above format.
{additional_context}
NODE ID : DESCRIPTION node_id : {node_description}
CHILDREN {children}

Table 7: Prompt for evaluating if an AND node's objective has been achieved.

You are a Global AND/OR Tree Update Agent revising an existing planning tree for a web-browsing task based on current available information so that the task is executed more efficiently. Do not change the ordering of the sub-plans.
Node definitions: Node Types <AND/ OR / Atomic> - AND Node: Represents an ordered list of logical subgoals required to achieve the node's objective. - OR Node: Represents alternative sub-strategies (which can be other AND/OR nodes) - Atomic Action: Single executable action strictly matching navigation_specifications described below. Node status indicators: - Unvisited nodes marked UNVISITED - Pruned nodes marked PRUNED - Completed nodes marked SUCCESS - Temporarily failed nodes marked FAIL
NAVIGATION_SPECIFICATIONS: <LIST OF BROWSER ACTIONS>
You are provided: -The root-level task description -Current AND/OR tree description -HTML DOM structure as the observation - Task progress summary - Notes summary: Summary of notes taken by the agent during the task
Your task is
to carefully analyze all the information provided and determine which nodes to prune and which nodes to update.
Apply changes in this strict order:
1. PRUNE nodes that are no longer relevant or are duplicates. 2. UPDATE node descriptions if intent is unchanged but content needs minor revision.
VERY IMPORTANT RULES: - You are only allowed to PRUNE or UPDATE nodes that have not been deleted , or pruned or marked succesful. - Do not add status of the node while updating the description. - Only use existing node IDs from the current tree; do not create new node IDs or subtrees. - Do not PRUNE important observations relevant to the task made in Atomic nodes with action 'note'. - Do NOT PRUNE children that are necessary for satisfying the parent node's objective. - Do not change node types. - Do not change the ordering of the sub-plans.
First reason about the update to the tree based on the information given (task description, item constraints, task progress summary, notes summary, current and/or tree description, observation) to you and then given your answer in the following format. Use the and/or tree description to ensure that you are not repeating any subgoals that have already been considered. Format each instruction on a new single line in the format given below (All children and description must be enclosed in «». Children should be colon (;) separated):
PRUNE [node_id] UPDATE [node_id] «Describe the node's new objective (subgoal/strategy/description of action)»
Example:
PRUNE [1.2] PRUNE [1.3] UPDATE [0.1.2] «type eggless cake in search bar» UPDATE [0.2] «Find an eggless cake recipe with over 60 votes and at least 4.5 star rating by examining search results»
Do not split the list across multiple lines.
ROOT-LEVEL TASK DESCRIPTION: {task_description}
{additional_context}
AND/OR TREE DESCRIPTION: {and_or_tree_description}
OBSERVATION: {observation}

Table 8: Prompt for global tree update.

You are a Web-browsing Agent specialized in web-browsing tasks. You have taken notes while completing a given task and now you need to generate an output for the task based on the notes you have collected. You are provided: - Task description - Sequence of notes taken during the execution of task - Planning tree representation of the task
Node definitions: Node Types <AND/ OR / Atomic> - AND Node: Represents an ordered list of logical subgoals required to achieve the node's objective. - OR Node: Represents alternative sub-strategies (which can be other AND/OR nodes) - Atomic Action: Single executable action strictly matching navigation_specifications described below. Node status indicators: - Unvisited nodes marked UNVISITED - Pruned nodes marked PRUNED - Completed nodes marked SUCCESS - Temporarily failed nodes marked FAIL
Your Task is to:
Give a detailed response that directly addresses the task description, using only the information from the notes and the planning tree presentation. Do not add any additional information or context that is not present in the notes. Even if the selected notes indicate that the task was not complete or certain information is missing or constraints are not met, provide the best possible response based on the available information. Do not make assumptions or inferences beyond what is explicitly stated in the notes.
VERY IMPORTANT RULES: - Do not assume any details or context that is not explicitly mentioned in the notes.
Output Format:
Begin by analyzing the notes, then give your detailed final response in the following format.:
Task Response: «Your detailed response to the task based on the notes. If the task was not completed, this may include incomplete information or recommendations that do not meet the constraints or requirements, if applicable. If the task was completed successfully, ensure all required details are specified. Do not skip any information that is required by the task.»
Example: Task Response: «I successfully found a recipe for an eggless chocolate cake with over 60 votes and a rating of 4.5 stars. The recipe includes ingredients such as flour, sugar, cocoa powder, and baking soda. I also noted that the recipe requires 30 minutes of preparation time....»
TASK DESCRIPTION: {task_description}
NOTES TAKEN DURING TASK EXECUTION: {notes}
{additional_context}

Table 9: Prompt for generating final task response from notes taken during execution.

fine-tuning generalized web agents using high-quality trajectories with dense rewards remains an underexplored research direction.

D Fine-tuning the Subplan Evaluator via Direct Preference Optimization

To train a subplan evaluator capable of ranking high-level plans, we utilize the *Direct Preference Optimization (DPO)* algorithm [Rafailov et al., 2024], which is designed to fine-tune policies based on preference comparisons rather than explicit ground-truth outputs. DPO is especially useful in scenarios where only relative preferences over candidate responses are available, rather than supervised labels.

In our setting, the DPO algorithm is applied to optimize a reward model π_r that can distinguish between good and bad subplans, conditioned on a task description and the initial observation. DPO learns from a dataset of preference tuples (x, y^+, y^-), where x denotes the context (in our case, task and initial observation), y^+ is the preferred subplan, and y^- is a less preferred or incorrect subplan.

To construct such a dataset, we begin with the NNetNav dataset [Murty et al., 2025], which contains successful task execution trajectories from real-world web environments. Each trajectory is

You are a Context Summarization and Critiquing Agent for web-browsing tasks.
Your job is to maintain an accurate, up-to-date understanding of task progress by analyzing:
- Task description
- Item-level constraints (if any)
- Task progress summary (if provided)
- Observation history (if provided)
- Action history (if provided)
- Notes summary: summary of notes taken by the agent so far (if provided)
- Current observation (HTML DOM)
INSTRUCTIONS:
0. Do not infer any details or make assumptions. 1. Analyze all available inputs, especially the current HTML DOM.
2. Ensure all summaries remain aligned with the main task objective.
3. Do not omit any information that may influence decisions or navigation.
4. Make summaries detailed, well-structured, and actionable.
5. Use the format below exactly.
6. Enclose the response to each section in double angle brackets: ‘« »‘.
Begin by analyzing the given context in detail. Then given your response stricly in the following format. Your output must ensure continuity, preserve all essential details, and contribute to successful task completion.
RESPONSE FORMAT:
OBSERVATION SUMMARY
«Describe the information from the CURRENT OBSERVATION. Emphasize elements and features that are relevant or potentially useful for fulfilling the task objective. Include all important detail.»
OBSERVATION HIGHLIGHTS
«Single list of integer element IDs (i.e., [123, 8765, 345]) from the current DOM that are relevant for interaction or for revisiting this step later. Only include elements from the current page. Sort by relevance, with the most important listed first.»
TASK PROGRESS
«Analyze the task progress using the task description, item-level constraints, past progress summaries, observation history, action history, and notes. First, summarize the key high-level steps the agent has actually taken toward completing the goal, focusing only on actions that were executed. Next, evaluate whether each explicitly stated task requirement or constraint has been satisfied—if the task involves item selection or recommendation, assess each item individually for compliance. Do not assume or infer unstated constraints. Finally, diagnose why the task has not yet been completed: identify any unsatisfied constraints, incorrect actions, or gaps in execution. Conclude with key takeaways or insights from the agent’s exploration that are relevant to achieving successful task completion.»
TASK FEEDBACK
«Based on the task progress summary, provide an outline of what the agent should focus on next to successfully complete the task. (2 sentences)»
Example:
OBSERVATION SUMMARY «The page shows search results for "iPhone 12 Pro Blue 128GB" on Amazon. The first relevant listing is an Apple iPhone 12 Pro, 128GB in Pacific Blue (Renewed) for $314.39. Multiple other iPhone models are also shown including iPhone 12, 13, and 14 in various colors and storage configurations....»
OBSERVATION HIGHLIGHTS
«[6028, 6033, 6042, 7204, 9277, 9280, 7242]»
NEW NOTES
«Refer to the task description and constraints, and identify all new information that can be used to completing the task. This should support future reference and continuity. Do not assume any information or make up new details. All details must be grounded in observation and actions. »
TASK PROGRESS
«The agent has successfully navigated to an Amazon search results page and identified a relevant product listing matching the task requirements. Specifically, the agent located an Apple iPhone 12 Pro with 128GB of storage in Pacific Blue, listed for $314.39 in renewed condition and fully unlocked. This matches the explicitly stated constraints for model, color, and storage capacity. The observation and notes confirm that the identified product has a strong customer rating (4.1 out of 5 from over 12,000 ratings), further supporting its relevance. All explicitly defined task constraints appear to be satisfied: model (iPhone 12 Pro), storage (128GB), color (Pacific Blue), fully unlocked, and condition (renewed). However, the task has not yet been completed. The agent has not taken action to click on the product listing, verify the full product details on the product page, or add the item to the cart. These remaining steps are essential for completing the task, as product details may vary across pages. The main takeaway is that the search and identification phase has been completed successfully, and the agent should now focus on verifying the product and adding it to the cart to fulfill the task requirements.»
TASK FEEDBACK
«The next step should be to click on the product listing to verify all details on the product page and then add it to the cart. The agent should ensure the exact model and specifications are confirmed before proceeding with the purchase. Focus on completing the final step of adding the item to cart.»
Do not output anything outside of the specified format.
TASK DESCRIPTION:
{task_description}
{additional_context}
CURRENT OBSERVATION:
{observation}

Table 10: Prompt for generating observation summary and contex summary.

You are an advanced web-browsing agent that solves web-browsing related tasks.
Your job is to generate notes base on current observation as well as formulate a response to the task query based on the notes and available information.
You are given:
- Task description
- Item-level constraints (if any)
- Task progress summary (if provided)
- Action history (if provided)
- Notes: Notes taken so far (if provided)
- Current observation (HTML DOM)
INSTRUCTIONS:
0. Do not infer any details or make assumptions.
1. Analyze all available inputs, especially the current HTML DOM.
3. Do not omit any information that may influence decisions or navigation.
5. Use the format below exactly.
6. Enclose the response to each section in double angle brackets: '« »'.
Begin by analyzing the context in detail. Given your final answer strictly in the following format.
RESPONSE FORMAT:
NEW NOTES
«Refer to the task description and constraints, and identify all new information that can be used to completing the task. This should support future reference and continuity. Do not assume any information or make up new details. All details must be grounded in observation. Do not skip any important notes relevant to the task. »
TASK RESPONSE
«Give a detailed response (you may repeat information from the above notes) that directly addresses the task description and constraints (if any), using only the information from the previous notes,new notes, and the action history. Do not add any additional information or context that is not present in the available information. Even if the selected notes indicate that the task was not complete or certain information is missing or constraints are not met, provide the best possible response based on the available information. Do not make assumptions or inferences beyond what is explicitly stated in the notes.»
Example:
NEW NOTES
«Found relevant iPhone 12 Pro listing matching requirements: - Model: iPhone 12 Pro - Storage: 128GB - Color: Pacific Blue - Price: $314.39 - Condition: Renewed - Fully Unlocked - Rating: 4.1/5 stars from 12,669 ratings»
TASK RESPONSE
«Based on the task requirements, I have found an Apple iPhone 12 Pro with 128GB of storage in Pacific Blue, fully unlocked, in renewed condition, and priced at $314.39. The listing has a solid customer rating of 4.1 out of 5 from 12,669 reviews.»
Do not output anything outside of the specified format.
TASK DESCRIPTION: {task_description}
{additional_context}
PREVIOUS NOTES: {notes}
CURRENT OBSERVATION: {observation}

Table 11: Prompt for generating notes summary.

represented as:

$$\mathcal{D} = \left\{ \left(T_j, \{(o_i^j, a_i^j)\}_{i=1}^{L_j} \right) \right\}_{j=1}^{M}$$

where T_j is the natural language description of the j-th task, $o_i^j \in \mathcal{O}$ is the observation at step i of trajectory j, $a_i^j \in \mathcal{A}$ is the atomic action taken at that step, L_j is the length of the trajectory, and M is the total number of trajectories.

From each successful trajectory, we extract a high-level plan $p_j = \{g_1, g_2, \ldots, g_n\}$ consisting of subgoals that were necessary and sufficient for completing the task. These subgoals are obtained by prompting a large language model with the task description T_j, the initial observation o_1^j, and the complete action sequence $\{a_i^j\}_{i=1}^{L_j}$.

Since the agent only observes o_1^j at the time of planning, we summarize this observation into a natural language form O_j, and define the input context as $x_j = (O_j, T_j)$.

To generate negative examples, we define a set of perturbation rules that simulate common subgoal planning errors observed in real trajectories. For each good plan p_j, we sample a rule and apply it using a language model to produce a corrupted plan $p_{j,l}^-$, where $l = 1, \ldots, K$. This gives us K negative plans per positive plan, resulting in a preference dataset of the form:

$$\mathcal{D}_{\text{pref}} = \left\{ (x_j, p_j, \{p_{j,l}^-\}_{l=1}^{K}) \right\}_{j=1}^{M}$$

where $x_j = (O_j, T_j)$ is the input context, p_j is the preferred (good) subplan, and $p_{j,l}^-$ are the corresponding perturbed (bad) subplans.

Learning with DPO. The DPO algorithm fine-tunes a policy $\pi_r(y \mid x)$ using preference comparisons. The reward function associated with a candidate subplan y under context x is defined as:

You are an advanced web-browsing agent. Your task is to maintain a table of candidate items by parsing the current webpage's HTML DOM and applying 'ADD', 'UPDATE', or 'DELETE' actions. Your goal is to keep the table accurate, up to date, and free of duplicates, based only on information explicitly visible in the DOM.
You are given:
-HTML DOM of the current webpage
- task description
- item-level constraints
VERY IMPORTANT
- Record all important details that are relevant to the task and can be used for completing the task. Include any additional relevant details.
- Only include keys whose values are **clearly visible** in the HTML DOM. Record the exact value corresponding to each key.
- Keys must match attributes listed under ITEMS AND CONSTRAINTS.
IMPORTANT RULES
Use this table to keep track of potential candidate solutions that can be used in the future for task completion or if search fails.
1. Matching and Deduplication
- Before using 'ADD', check whether the item already exists in the candidate table.
- An item is considered a duplicate if all its visible fields match an existing entry (even if the ID differs).
- If a match is found:
- Use 'UPDATE' with the existing ID.
- Only include fields that are newly visible or changed.
- Do **not** use 'ADD' for duplicates.
2. Field Visibility - Only include fields with values explicitly visible in the HTML DOM.
- Do **not** guess or infer missing values.
- Add exact values instead of simply checking if constraints is satisfied.
- Do **not** include '"unknown"' or empty-string values for missing fields.
- Omit any field that is not explicitly visible.
3. Constraint Checking
3.1 Basic Constraint Handling - For each item, evaluate which required constraints are satisfied.
- If any are violated or missing:
- Add 'constraints_not_met' listing the missing/invalid fields (e.g., 'lift range, tray type')
- Set 'status: uncertain'
- Add a brief 'comment' (e.g., "missing lift range")
3.2 Add Only When Most Constraints Are Met - Only use 'ADD' if the item satisfies **most** required constraints (at least 60%).
- If fewer than half are satisfied: - Do **not** add the item.
- If it already exists, consider using 'UPDATE' or 'DELETE'.
4. When to Use 'UPDATE'
Only use 'UPDATE' when **at least one of the following is true**:
- A previously missing field has now been found (and was not already present)
- The item now meets all required constraints and can be marked 'status: complete'
Do **not** update items:
- If nothing has changed
- If the values are already present and correct
- If the item is already marked 'deleted'
5. Deletion - Use 'DELETE' only if the item is **clearly invalid** — for example:
- It is irrelevant
- It is severely incomplete (missing most required fields)
- It does not satisfy constraints and is not fixable
- Never issue 'UPDATE' for an item that has already been deleted.
6. ID Assignment
- When using 'ADD', assign a unique ID not already present in the table (e.g., 'S102', 'CH014').
7. Item Validity
- The 'item' name must match a known type defined under ITEMS AND CONSTRAINTS.
- Skip any item with an invalid or unrecognized type.
RESPONSE FORMAT
- Return only valid 'ADD', 'UPDATE', or 'DELETE' actions — one per line.
- Do **not** return the HTML DOM or candidate table.
- Do **not** include explanation or commentary.
- If no action is needed, return an empty string.
Your response must strictly follow the following format.
Use one or more of the following actions, one per line:
ADD item:ID:UniqueID; key1:value1; key2:value2; ...
UPDATE item:ExistingID key1:value1; key2:value2; ...
DELETE item [ExistingID]
Note: item must match exactly to the items in item-level constraints.
While adding new items, always include the name or the title of the item.
EXAMPLES
ADD standing desk:ID:S102; Title:ErgoRise Desk; Price:$299.99; surface finish:Matte; status:uncertain; comment:"missing tray type and lift range"
UPDATE standing desk:S101 lift range:"24×48″"; status:complete; comment:"lift range confirmed"
DELETE standing desk [S099]
TASK DESCRIPTION {task_description}
ITEMS AND CONSTRAINTS {items_and_constraints}
CURRENT CANDIDATE TABLES {current_table}
WEB PAGE CONTENT {current_observation}

Table 12: Prompt for extracting candidate items to add to structured memory

You are a web-browsing assistant. Given a user query, extract a list of items to retrieve and, for each item, list all relevant item-specific constraints such as brand, price, rating, features, delivery time, etc.
Include any other item attributes explicitly or implicitly mentioned in the query that affect the desirability, quality, or functionality of the item (e.g., surface finish, memory presets, power type, compatibility). Treat these as additional constraints.
Include all intrinsic item-specific constraints, even if mentioned implicitly or as part of comparison criteria (e.g., "differ in brand or surface finish" implies that brand and surface finish are relevant constraints). Treat such attributes as constraints to be listed.
Ignore any source-specific constraints (e.g., "from [a specific website]") and general task-level instructions (e.g., "compare prices", "top three results"). Only include intrinsic item-specific constraints.
Only include items to retrieve—ignore any actions to perform. Ensure each item listed is unique. If there are no items to retrieve, return an empty list.
Output your response strictly in the following JSON format:
"ITEMS": ["item": "<description of item>", "constraints": ["<constraint1>", "<constraint2>", "..."] , ...]
Do not include any additional explanation. Always follow this format exactly.
TASK DESCRIPTION
{task_description}
ITEMS AND CONSTRAINTS
{items_and_constraints}
CURRENT CANDIDATE TABLES
{current_table}
WEB PAGE CONTENT
{current_observation}

Table 13: Prompt for extracting item-level constraints and attributes from task description.

You are a validation assistant for a candidate item table.
Your task is to verify whether each item satisfies the required constraints for its item type. If any constraint is violated or missing, issue an 'UPDATE' or 'DELETE' command to adjust the table accordingly.
INSTRUCTIONS
1. For each item:
- Issue an 'UPDATE' only if all of the following are true:
- The item is not marked as deleted
- One or more constraints are violated or missing
- The current values of 'constraints_not_met', 'status', or 'comment' are incorrect, incomplete, or missing
- Use the following format:
"' UPDATE item:ID constraints_not_met: <key1> <key2> ..., status: uncertain or complete; comment: "brief explanation" "'
- You are only allowed to update the following fields: - 'constraints_not_met' - 'status' - 'comment'
- Do not issue duplicate updates. Skip updates if no changes are needed.
- Issue a 'DELETE' only if:
- The item is marked as complete but fails to meet any constraint
- Or the item is mostly complete but violates most required constraints
- Use this format: "' DELETE item [ID] "'
- Do not delete items that are mostly incomplete or missing most of the required fields. These should be updated instead.
2. Item Validity - The 'item' in every 'UPDATE' or 'DELETE' command must exactly match a valid item type defined under 'ITEMS AND CONSTRAINTS'.
- Skip any action for items with invalid or unrecognized types.
3. If the item satisfies all constraints and is already marked 'status: complete', or if the item is incomplete but does not require any changes, return nothing.
RESPONSE FORMAT
- Return only 'UPDATE' and 'DELETE' commands, one per line.
- Do not return the full item rows or any explanation.
- If no action is needed, return an empty string.
- Use one or more of the following actions, one per line.
- Note: item must match exactly to the items in item-level constraints.
Your response must strictly follow the following format.
Use one or more of the following actions, one per line:
UPDATE item:ExistingID key1:value1; key2:value2; ...
DELETE item [ExistingID]
EXAMPLES
UPDATE standing desk:SD002 constraints_not_met: <lift range> <tray type>; status: uncertain; comment: "Missing lift range and tray type"
DELETE standing desk [SD010]
TASK DESCRIPTION
{task_description}
ITEMS AND CONSTRAINTS
{items_and_constraints}
CANDIDATE TABLE:
{current_table}

Table 14: Prompt for updating structured memory

You are an AND/OR Tree Repairer Agent specialized in web-browsing tasks. Your job is to repair a node in an AND/OR planning tree derived from HTML DOM structure observations.
Node definitions:
Node Types <AND/ OR / Atomic>
- AND Node: Represents an ordered list of logical subgoals required to achieve the node's objective.
- OR Node: Represents alternative sub-strategies (which can be other AND/OR nodes)
- Atomic Action: Single executable action strictly matching navigation_specifications described below.
Node status indicators:
- Unvisited nodes marked UNVISITED
- Pruned nodes marked PRUNED
- Completed nodes marked SUCCESS
- Temporarily failed nodes marked FAIL
NAVIGATION_SPECIFICATIONS:
<LIST OF BROWSER ACTIONS>
You are provided:
- root-level task description
- task constraints at item level - HTML DOM structure as the observation - node_id and desription of the node to analyze - A list of its children: each with an ID, description, and status (SUCCESS, VISITED, UNVISITED, or PRUNED) - A successor node with id and description, if present - Task progress summary of the task - Notes summary of the task : Notes taken by the agent so far
Your task:
Your task is to review all the information provided including reasoning for repair (if available) and use it to repair the node. Make sure you prune infeasible subgoals (which have not been marked successful or pruned) and add subgoals that are necessary for completing the node's objective:
Repair the node by pruning or adding children to the node: Remove old infeasible children and add new children that were not previously used.
VERY IMPORTANT: -Assess node repairability using node type, child status, and remaining unmet goals. -Add only new, untried children , not already present or marked SUCCESS or PRUNED. These children will be appended to existing children. -Each child must be minimal, concrete, and justified by the current DOM. -For OR nodes: Suggest only new, obvious alternatives from the DOM. -For AND nodes: Suggest only missing or incomplete subgoals in correct order. -Do NOT PRUNE children that are necessary for satisfying the parent node's objective.
First reason about the repair and then give the final answer in the following format (Enclose all values in «» on a single line, colon (;) separated. Ordering: Prune first, then add children):
PRUNE [node_id of child] ADD [repair_node_id] repair_node_type : «1. First added subgoal; 2. Second added subgoal; 3. Additional added subgoals as necessary» OR ADD [node_id] node_type : «1. Added alternative strategy 1 (score: 0.x); 2. Added alternative strategy 2 (score: 0.y); Added additional alternatives as necessary» Reasoning «Reason for the nodes pruned and added.»
Example 1: PRUNE [1246] PRUNE [1245] ADD [1244] AND : «1. Go north for 5 meters; 2. Turn left at the intersection; 3. Walk for 10 minutes»> OR ADD [1244] OR : «1. Open the door (score: 0.9); 2. Pick up the key (score: 0.8)»
Do not re-add processed or pruned children. Note that you can only prune children of the node to repair and add children to the node to repair. Added children will be appended to existing successful children. Respond using only the above. No extra text. Valid node_types are AND/OR/Atomic .
ROOT-LEVEL TASK DESCRIPTION: {task_description}
{additional_context}
OBSERVATION: {observation}
NODE ID : Description {node_id} {node_type} : {node_description}
NODE_CHILDREN_REPRESENTATION: children
REASON FOR REPAIR reason_for_repair
LOCAL TREE INFORMATION: {local_tree_info}

Table 15: Prompt for Node Repair

$$r(x,y) = \beta \log \frac{\pi_r(y \mid x)}{\pi_{\text{ref}}(y \mid x)} \; + \; \beta \log Z(x),$$
$$\text{where Z(.) is a partition function defined as } Z(x) = \sum_y \pi_{\text{ref}}(y \mid x) \exp\Big(\tfrac{1}{\beta}\, r(x,y)\Big). \tag{1}$$

Here, π_r is the reward policy being trained and π_{ref} is a fixed reference policy. In our case, π_{ref} is the initial supervised fine-tuned (SFT) model, denoted π_{SFT}, which is trained only on the good plans extracted from the successful trajectories. Thus,

$$r(x,y) = \beta \log \frac{\pi_r(y \mid x)}{\pi_{\text{SFT}}(y \mid x)} \; + \; \beta \log Z(x).$$

The parameter $\beta \in \mathbb{R}^+$ controls the deviation from the reference policy π_{SFT}; smaller values of β encourage the reward model to stay closer to the reference, while larger values allow greater deviation in favor of stronger preference alignment. This reward encourages π_r to assign higher probability to plans that are preferred over those rejected, while regularizing against drift from π_{SFT}.

Given a preference pair $(p_j, p_{j,l}^-)$ under context x_j, the DPO loss is given by:

$$\mathcal{L}_{\text{DPO}} = -\log\left(\frac{\exp(r(x_j,p_j))}{\exp(r(x_j,p_j)) + \exp(r(x_j,p_{j,l}^-))}\right).$$

This objective is minimized across all preference pairs in $\mathcal{D}_{\text{pref}}$ to obtain the final trained reward policy π_r.

After training, the policy π_r serves as a reward model capable of scoring subplans. At inference time, given a task description T and an initial observation O, we construct a context $x = (O, T)$ and

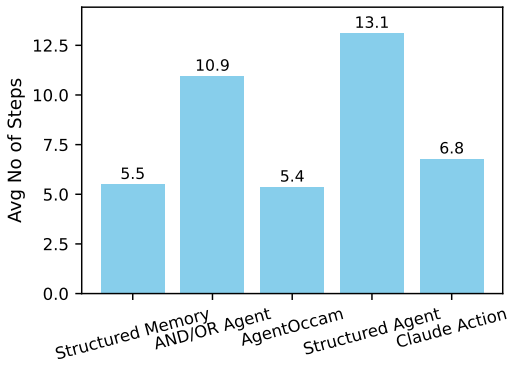

Figure 5: Average number of steps taken by the agents on Complex Shopping tasks using *Claude 3.5* as the backbone model.

evaluate a set of candidate subplans $\{p_1, \ldots, p_P\}$ using the DPO reward function. The best subplan is selected as:

$$p^* = \arg \max_{p \in \{p_1, \ldots, p_P\}} r(x, p) = \arg \max_{p \in \{p_1, \ldots, p_P\}} \beta \log \frac{\pi_r(p \mid x)}{\pi_{\text{SFT}}(p \mid x)}.$$

This allows us to rank and select subplans based on their relative preference score, as learned from comparisons during training.

E Experimental Details

Amazon Shopping Tasks

- Recommend 3 standing desks under $350 that support lift ranges from $\leq$ 24" to $\geq$ 48" and include a cable management tray. Each recommendation should differ in brand or surface finish. Do not add to cart–list lift range, tray type, and price.
- Recommend 3 pressure cookers under $100 with stainless steel inner pots and yogurt mode. Each must be from a different brand. Do not add to cart–list material, features, rating, and price.
- Recommend 3 wireless printers under $150 that support both AirPrint and borderless 4x6 photo printing. Do not add to cart–list brand, print support, and price.
- Recommend 3 mechanical keyboards under $400 with hot-swappable keys. Ensure variety in switch type or layout (e.g., 75%, TKL). Do not add to cart–list hot-swap support, switch type, and price.
- Recommend 3 laptop backpacks under $200 that support 17-inch laptops, include a padded sleeve, and have a hidden anti-theft pocket. Ensure variation in material, port support, or design. Do not add to cart–list capacity, features, and price.
- Recommend 3 noise-canceling headphones under $300 with multi-device pairing and passive listening support. Ensure brand diversity. Do not add to cart–list battery specs, pairing support, and price.
- Recommend 3 air purifiers under $350 with washable pre-filters and a night mode that disables display lights. Do not add to cart–list pre-filter type, noise level, and price.
- Recommend 3 external SSDs (1TB) under $120 that use USB-C and offer hardware encryption. Ensure diversity in brand or ruggedness. Do not add to cart–list encryption support, interface, and price.
- Recommend 3 espresso machines under $450 that include a milk frother and removable water tank. Ensure different frother types or build designs. Do not add to cart–list milk system, tank size, and price.

- Recommend 3 portable projectors under $300 with 1080p native resolution, tripod screw mount, and built-in Bluetooth speakers. Ensure model diversity. Do not add to cart–list resolution, mounting, audio features, and price.
- Find the cheapest projector and screen set under $450: (1) portable projector with native 1080p, Bluetooth 5.0+, keystone correction, and 300+ ANSI lumens; (2) 70–100 inch outdoor screen. Recommend 3 combinations varying by projector brand. Choose the set with the highest total reviews. List model names, Bluetooth version, brightness, and total price.
- Find the cheapest pair under $350 that includes: (1) over-ear wireless ANC headphones with 40h+ battery life and USB-C charging; (2) premium Bluetooth 5.0+ speaker with stereo pairing and IPX5 water resistance. Both must be rated 4.3+ stars. Recommend 3 brand-distinct combos. List battery specs, pairing and water-resistance features, rating, and total cost.
- Recommend 3 lightweight work kits under $400 with fast delivery: (1) adjustable aluminum laptop stand supporting 10kg+; (2) rechargeable wireless keyboard + mouse; (3) USB-C hub with HDMI, SD, and Ethernet. Choose the combo with the fastest shipping (2 days or less). List ports, delivery times, and prices.
- Under $450, find the most-reviewed premium travel set: (1) 17 inch anti-theft backpack with TSA lock; (2) 8-piece compression packing cube set made from water-resistant fabric; (3) digital luggage scale with auto-off, tare, and backlight. All rated 4.3+ stars. Recommend most-reviewed option. List features, review counts, and total price.
- Recommend the cheapest high-end ergonomic combo under $900: (1) dual-motor standing desk (dual 275+ lb capacity, cable tray, programmable presets); (2) office chair with adjustable headrest, and lumbar support. Both must be rated 4.5+ stars and have 500+ reviews. List brands, specs, and combined price.
- Find the fastest-delivery smart home bundle under $500: (1) smart speaker with built-in voice assistant and premium audio; (2) smart bulb pack with color plus tuneable white and 10K+ hour lifespan; (3) smart plug with energy monitoring and USB port. All must be rated 4.5+ stars and offer two-day delivery. List assistant, bulb specs, plug features, delivery ETA, and price.
- Recommend 3 pro-level chef's prep kits under $400: (1) 8" stainless chef's knife with full tang and thermo-transfer handle; (2) digital food thermometer with $\pm 0.1^{\circ}$C accuracy, fast-read (<5s), and auto-off; (3) bamboo cutting board set with juice groove and non-slip feet. Choose highest-rated combination. List specs, ratings, and price.
- Find the most-reviewed outdoor adventure set under $500 each: (1) waterproof hiking backpack (30–40L) with rain cover; (2) rechargeable headlamp (500+ lumens) with adjustable beam; (3) vacuum-insulated stainless steel bottle (32 oz) with sweat-proof exterior. All rated 4.5+ stars. List capacities, feature specs, review counts, and total price.
- Recommend 3 advanced artist kits under $250: (1) LED desk lamp with CRI$>$ 90 and adjustable 3000K–6500K color; (2) sketchbook A4 with 200gsm acid-free paper; (3) graphite pencil set including 2H–8B and charcoal. Kits must differ by lamp or sketchbook brand. List specs, brand, review counts, and price.
- Find the fastest-delivery premium pet care bundle under $250: (1) hands-free dog leash with reflective stitching, padded waist belt, and shock absorber; (2) collapsible BPA-free silicone travel bowl; (3) no-pull harness with five-point control and breathable mesh. All rated 4.5+ stars with two-day delivery. List materials, feature highlights, shipping ETA, review counts, and combined price.
- Recommend 2 cordless stick vacuums under $200 with detachable batteries, HEPA filtration, at least 25 minutes runtime, and listed runtime and battery replacement ease.
- Find 3 premium gaming chairs under $300 with adjustable lumbar support, at least 150-degree recline, verified user weight support of at least 300 lbs, and listed weight capacity and user review rating.
- List 2 portable espresso makers under $100 compatible with Nespresso capsules, at least 18 bar pressure, BPA-free certification, and clearly state compatibility, pressure, and BPA-free status. If criteria aren't met, explain the closest match.

- Recommend 3 noise-cancelling earbuds under $120 with at least 24-hour total playtime, IPX5 or higher waterproof rating, transparency mode, and clearly state sound quality ratings and battery life duration.
- Find 2 smart thermostats under $150 compatible with Alexa and Google Assistant, supporting multi-zone control, energy-saving certification, and clearly state assistant compatibility, multi-zone capability, and certification details. If unavailable, suggest best alternatives with limitations.
- List 3 robot vacuums under $250 with mapping capability, no-go zones, voice assistant support, runtime, and bin capacity. Identify models that fulfill most criteria if any feature is missing.
- Recommend 2 air fryers under $100 with at least five-quart capacity, dishwasher-safe basket, preset cooking functions, and listed cooking presets and ease of cleaning. Highlight trade-offs between price, capacity, and features if necessary.
- Find 3 fitness trackers under $80 with continuous SpO2 monitoring, swim-proof rating of at least 50 meters, sleep-tracking, battery life, and additional health monitoring features clearly listed.
- List 2 electric toothbrushes under $60 with pressure sensors, at least 30-day battery life, ADA acceptance, charging type, and brush head replacement availability. Recommend based on user reviews and clinical backing if ADA acceptance is missing.
- Recommend 3 webcam models under $70 with autofocus, at least 1080p resolution, 60 frames per second streaming, built-in privacy cover, and microphone quality. Clearly identify compromises on frame rate or privacy cover if necessary.
- Recommend 2 wireless routers under $200 with Wi-Fi 6 support, dual-band, gigabit Ethernet ports, and listed max coverage area. If no routers meet all, choose closest and note missing attribute.
- Find 3 kitchen stand mixers under $300 with at least 10 speeds, 5-quart bowl, and metal construction. List speed count, bowl size, and material. If fewer than three, include models that miss one requirement and indicate which.
- List 2 Bluetooth speakers under $150 with waterproof rating of IPX7, at least 12-hour battery life, and built-in voice assistant support. State battery life and assistant type. If criteria aren't met, explain closest fit.
- Recommend 3 DSLR-style mirrorless cameras under $700 with interchangeable lenses, 4K video, and in-body image stabilization. Clearly state sensor resolution, video spec, and stabilization type. If none, suggest best trade-offs.
- Find 2 cordless electric lawn mowers under $400 with at least 45 minute runtime, 20-inch deck, and mulching capability. List runtime, deck size, and whether mulching kit included.
- List 3 external SSDs under $150 with USB-C connection, at least 1 TB capacity, and read speed over 1000 MB/s. Provide capacity, interface, and read speed. If fewer than three, note closest specs.
- Recommend 2 smartwatches under $200 with built-in GPS, NFC payments, and ECG or heart-rate variability tracking. State GPS, NFC, and health feature. If none have ECG, list HRV instead.
- Find 3 midsize camping tents under $200 with capacity for at least four people, full rainfly, and tent weight under 15 pounds. List capacity, rainfly type, and weight. If criteria not fully met, show closest option.
- List 2 home security cameras under $100 with 1080p resolution, night vision, two-way audio, and local storage option. Provide each feature's status. If no local storage, note cloud-only limitation.
- Recommend 3 pair of running shoes under $120 with carbon-fiber plate or equivalent propulsion tech, neutral support, and weight under 10 ounces. State plate tech, support type, and weight. If none, note closest feature set.

- Recommend 2 DSLR lenses under $500 for wildlife photography that have at least 300mm focal length, image stabilization, and autofocus under 0.5s. List focal length, stabilization type, and measured autofocus time. If none, suggest closest alternatives and note compromises.
- Find 3 camping stoves under $150 that support propane and butane, boil one liter of water in under four minutes, and have built-in wind protection. Provide fuel type compatibility, boil time, and wind guard design details. If fewer than three meet all, include near matches with missing features.
- List 2 insulated tumblers under $40 with 30-hour cold retention, 12-hour hot retention, and dishwasher-safe lid. State retention times, lid type, and size. If criteria are missing, recommend closest and explain trade-offs.
- Recommend 3 external monitors under $300 with at least 27-inch size, IPS panel, 75 Hz refresh rate, and USB-C power delivery. Provide screen size, panel type, refresh rate, and wattage delivered via USB-C. If power delivery is absent, note fallback features.
- Find 2 portable power stations under $400 with AC outlets, solar charging support, and at least 500 Wh capacity. List AC output wattage, solar input type, and capacity. If solar charging not supported, mention alternative recharge methods.
- List 3 noise-monitoring baby monitors under $200 with temperature display, lullaby/music playback, and two-way talk. Provide screen size (or app), temperature reporting, and audio features. If lullaby feature is missing, note which are closest.
- Recommend 2 countertop ice makers under $250 that produce at least 26 lbs of ice per day, have self-clean function, and use bullet-shaped ice. List daily output, cleaning cycle, and ice type. If none match exactly, propose closest and trade-offs.
- Find 3 inflatable paddle boards under $700 with maximum load of at least 300 lbs, included pump, and thickness of at least 6 inches. Provide max load, pump type, and board thickness. If load capacity slightly lower, note it.
- List 2 smart jump ropes under $100 with integrated fitness tracking, Bluetooth app syncing, and rechargeable battery. State tracking metrics, app availability, and battery runtime. If rechargeable is not available, recommend suitable near match.
- Recommend 3 gaming keyboards under $150 with per-key RGB lighting, hot-swappable switches, and dedicated macro keys. Provide switch type, lighting software, and macro implementation details. If hot-swap is not present, note it.
- Recommend 3 sulfate-free shampoos under $25 that are color-safe, contain at least 2% argan oil, and have a pH between 5 and 6. List ingredient percentages, color-safe claims, and pH value. If pH is not listed, note this clearly.
- Find 2 paraben-free retinol serums under $50 with at least 0.5% retinol, added vitamin C, and cruelty-free certification. Clearly state retinol and vitamin C concentrations, and certification status.
- List 3 mineral sunscreens under $30 that are zinc-oxide based, at least SPF 30, reef-safe (no oxybenzone/octinoxate), and water-resistant for at least 80 minutes. Provide SPF, zinc-oxide percentage, and water resistance time.
- Recommend 2 fragrance-free facial moisturizers under $40 with hyaluronic acid, non-comedogenic, and dermatologist-tested. List hyaluronic acid percentage, comedogenic rating, and dermatologist testing claims.
- Find 3 sulfate-free cleansing oils under $35 with at least two plant oils, vitamin E, and eco-cert organic certification. Clearly state oil types, vitamin E content, and certification status.
- List 2 paraben-free body lotions under $20 with at least 10% shea butter, fast-absorbing formula, and allergy-tested status. Provide shea butter percentage, absorption claim, and testing certification.
- Recommend 3 vegan lipsticks under $25 that are cruelty-free, provide at least six-hour wear, and include SPF 15. List wear time, SPF rating, and certification details. If SPF is absent, clearly state the trade-off.

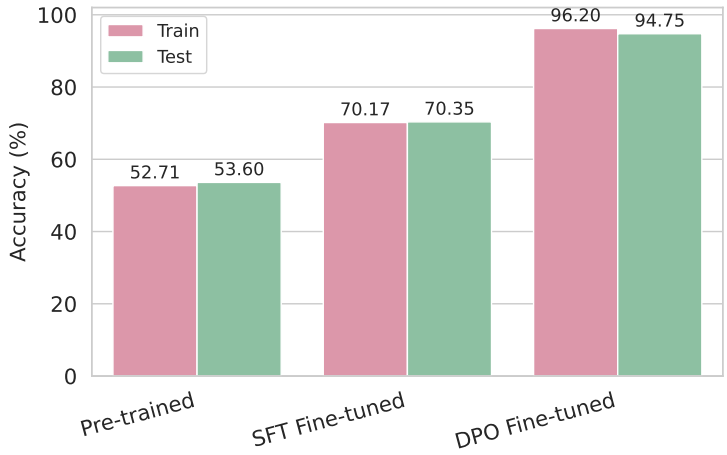

Figure 6: Classification accuracy of the Subplan Reward Model on the train and test sets, evaluated using the pre-trained, SFT fine-tuned, and DPO fine-tuned versions of the *unsloth/Meta-Llama-3.1-8B-Instruct* model.

- Find 2 alcohol-free facial toners under $30 with probiotics or niacinamide, non-pore-clogging, and a pH between 4 and 5. List active ingredients and pH; if pH is not listed, clearly note it.
- List 3 caffeine-infused eye creams under $35 that are fragrance-free, paraben-free, and claim to reduce puffiness within 15 minutes. State caffeine concentration, effectiveness claim timing, and ingredient exclusions.
- Recommend 2 multifunctional balm sticks under $30 that are petroleum-free, contain SPF, and can be used on lips, cheeks, and cuticles. Provide ingredient list, SPF rating, and specified usage areas. Clearly note if multifunction use is limited.

F Structured Agent Framework

F.1 Subplan Evaluator

For evaluating the **subplan evaluator**, we assess the accuracy of the learned reward model on a held-out test set of subplan preferences. This is treated as a binary classification task where the model must choose the preferred plan between each positive-negative plan pair. We report classification accuracy as the primary metric for this component.

Figure 6 evaluates a pre-trained `Llama-3.1-8B-Instruct` model as a subplan evaluator, comparing it to models fine-tuned using SFT and DPO on the test subplan preference dataset. Fine-tuning with SFT improves classification accuracy by approximately 17%, while subsequent fine-tuning with DPO provides an additional 24% gain, resulting in a total 41% improvement over the pre-trained model. These results highlight the effectiveness of our reward modeling approach.