

Improving Language Agents through BREW: Bootstrapping experientially-learned Environmental knowlEdge

Shashank Kirtania^{1*} Param Biyani² Priyanshu Gupta³ Yasharth Bajpai³
Roshni Iyer⁴ Sumit Gulwani³ Gustavo Soares³

¹University of Michigan, ²Independent, ³Microsoft, ⁴Apple.

Abstract

Large Language Model (LLM)-based agents are increasingly capable of complex, multi-step tasks such as GUI automation, tool use, and data manipulation, yet they cannot learn from experience: each new session rediscovers solutions from scratch. We introduce BREW (Bootstrapping experientially-learned Environmental knowlEdge), a framework that distills an agent’s past interaction trajectories into a structured, retrievable knowledge base (KB) of natural-language *recipes*, concept-level procedural documents that capture *what* to do, *when* it applies, and *what* to watch out for. Drawing on the principle of library learning from program synthesis, BREW decomposes agent memory into modular, concept-localized documents and formalizes KB construction as a state-space search problem. To navigate this space, we introduce Expand-and-Gather Monte Carlo Tree Search (EG-MCTS), a reward-guided algorithm that jointly optimizes recipe accuracy and retrievability across parallel, per-concept search trees. We further adapt hindsight relabeling to convert near-miss trajectories into positive demonstrations, surfacing latent agent competencies as reusable knowledge. On three domain-grounded benchmarks, OSWORLD, τ^2 -Bench, and Spread-SheetBench, BREW achieves 10–20% gains in task success and 10–15% fewer execution steps over base agents, while consistently outperforming existing memory-augmented baselines that can degrade below memoryless performance. The resulting KB is inspectable, modular, and extensible, providing a transparent and controllable substrate for agent optimization.

1 Introduction

Large Language Model (LLM) based agents are increasingly capable of interacting with complex environments like navigating GUIs, calling external tools, and manipulating structured data across a wide range of real-world tasks (Li, 2025; Qin et al., 2025; Jimenez et al., 2024; Yang et al., 2024; Anthropic, 2024; OpenAI, 2025). Yet these agents lack a fundamental ability that humans take for granted: learning from experience. When an agent encounters a familiar task in a new session, it rediscovers the solution from scratch, repeating the same wrong menu paths, redundant API calls, or failed spreadsheet manipulations it had already resolved before.

After exporting a document to PDF once in LibreOffice, a person does not memorize the raw click sequence; they internalize a *recipe*: the general pattern (File → Export as PDF), the conditions under which it applies (any Writer or Impress document), and the pitfalls to avoid (confirming the output opens correctly). Similarly, after handling a few customer return requests, a support agent learns the procedural structure: authenticate the user, verify the order status is delivered, collect all exchange items upfront because the step cannot be

*1,2,4 Work done at Microsoft.

Corresponding author: priyansgupta@microsoft.com
Code and other artifacts: aka.ms/brew-code

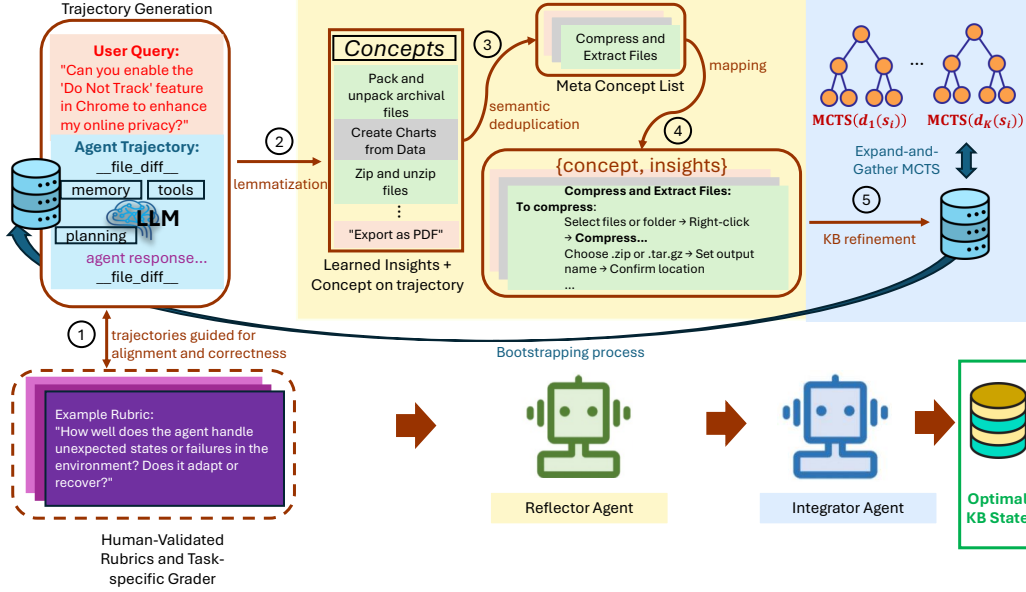


Figure 1: BREW architecture overview using examples from the OSWORLD dataset. Step 1 indicates trajectory generation with agent alignment to human-validated rubrics and correctness using a task-specific grader. Steps 2–4 indicate the Reflector Agent, which learns key concepts and insights from trajectories. Step 5 indicates the Integrator Agent, which integrates knowledge from the Reflector Agent to bootstrap the KB. We introduce Expand-and-Gather MCTS to find the best KB configuration by reward-guided search.

repeated, then execute. These recipes transfer across tasks within the same environment, compressing future problem-solving into a few reliable steps.

How should an agent acquire such recipes? Weight optimization (Schulman et al., 2017; Rafailov et al., 2024; Shao et al., 2024) bakes experiential knowledge into model parameters, producing opaque policies that resist inspection or modular update. Memory-augmented agents (Chhikara et al., 2025; Xu et al., 2025b; Gupta et al., 2024b; Agrawal et al., 2025) pursue explicit storage but at the wrong granularity, retaining either transient trajectory fragments or high-level reflections that lack the specificity to guide execution. A more principled template comes from library learning in program synthesis (Ellis et al., 2020), where agents refactor recurring subroutines into *explicit, modular objects* that live outside the model, inspectable, updatable, and composable, dramatically compressing the search space for future tasks. We adopt the same principle but observe that the abstractions real-world agents need are not code primitives: they are *natural-language recipes*, structured descriptions of what to do, when it applies, and what to watch out for, because the environments themselves (GUIs, tool APIs, conversational interfaces) are described and interacted with in language (see Section 2 for a detailed discussion).

We introduce BREW (**B**ootstrapping experientially-learned **e**nvironmental **k**nowledge), a framework that learns a recipe library from agent experience (Figure 1). BREW incrementally constructs and refines a knowledge base (KB), a structured collection of concept-level procedural documents in natural language, directly from an agent’s past interaction trajectories. Each document captures a reusable behavioral pattern at the right level of abstraction: more specific than a reflection (“be careful with file formats”) yet more general than a raw trajectory (the full click-by-click trace of one export). The KB serves as persistent, retrievable memory that the agent queries during future executions, compressing the search space the way a learned library compresses program synthesis. Because natural-language recipes lack the clean verification signals of compiled code, we formalize KB construction as a state-space search problem and introduce Expand-and-Gather Monte Carlo Tree Search

(EG-MCTS), a reward-guided algorithm that finds recipe formulations that are both accurate for downstream reasoning and retrievable at inference time. Our key contributions are:

- *Recipe learning from agent trajectories.* We propose a technique for distilling concept-level procedural recipes from agent interaction trajectories. Guided by rubrics and task-specific graders, this process partitions raw experience into modular, human-interpretable knowledge, the natural-language analog of library functions in program synthesis.
- *State-space search for recipe optimization.* We formalize KB refinement as a state search problem and introduce EG-MCTS, which explores the space of recipe formulations via parallel, reward-guided search across concept-level documents, addressing the noise and ambiguity inherent in optimizing natural-language abstractions.
- *Hindsight relabeling for broader recipe coverage.* We adapt hindsight relabeling (Andrychowicz et al., 2018) to expand the diversity of learned recipes, converting near-miss trajectories into positive demonstrations that reveal latent agent competencies as reusable knowledge.
- *State-of-the-art results.* On domain-grounded benchmarks including OSWORLD and τ^2 -Bench, BREW achieves 10–20% gains in task precision and 10–15% fewer execution steps, while maintaining memory and compute costs comparable to base LLMs.

2 Related Works

Agent Learning from Demonstrations The principle of *library learning* from program synthesis—where agents refactor recurring subroutines into reusable primitives that compress future search (Ellis et al., 2020)—motivates a growing body of work on structured skill extraction for LLM agents. Recent methods leverage LLMs to isolate reusable skills through interactive decomposition (Hashemzadeh et al., 2024), synthesize executable domain-specific functional abstractions (Khan et al., 2025), or learn in-prompt memory (Gupta et al., 2024b). However, these approaches rely on static decomposition or offline synthesis, and the code-level primitives they produce do not naturally capture procedural, environment-specific knowledge (e.g., that “Ctrl+F opens search in Nautilus”). BREW addresses both limitations: it iteratively refines natural-language *recipes* via rollout-generated insights and structured knowledge-base search (EG-MCTS). Unlike prompt-optimization techniques (Agrawal et al., 2025; Gupta et al., 2024b), the resulting recipes are stored as retrievable, extensible agent-memory knowledge bases.

Agentic Memory The concept of providing agents with controllable memory has a rich history (Littman, 1993). Memory mechanisms are attracting increasing attention (Packer et al., 2024; Wang et al., 2025; Xu et al., 2025a; Chhikara et al., 2025; Xu et al., 2025c; Hu et al., 2025), typically storing relevant context in structured formats such as graphs or trees for retrieval-augmented generation. While effective in their target sub-domains, these techniques fail to generalize across environments (Hu et al., 2025). More fundamentally, model weight optimization (Schulman et al., 2017; Rafailov et al., 2024; Shao et al., 2024) learns abstractions implicitly by baking them into parameters, producing opaque policies that resist inspection or modular update and couple tightly to training distributions. At the other extreme, memory-augmented agents such as Mem0 (Chhikara et al., 2025; Xu et al., 2025b) store transient trajectory fragments that vanish between episodes, while MetaReflection (Gupta et al., 2024b) and GEPA (Agrawal et al., 2025) embed high-level reflections that lack the specificity to guide execution.

State Based Explorations State-space search has been extensively used for exploration based learning (Silver et al., 2016; Liu et al., 2025). With the advent of prompt-tuned LLM systems, state space techniques are being actively explored in the community for text-based optimization (Gupta et al., 2024a; Wang et al., 2023; Novikov et al., 2025). Our technique builds upon this work and generalizes it to general-purpose agent memory learning.

3 BREW: Architecture

This section describes our proposed **B**ootstrapping **e**xpeRientially-learned **E**nvironmental **k**no**W**ledge technique, BREW, which constructs and iteratively refines a Memory KB using trajectory insights guided by human-validated general-purpose agent behavior metrics, task-specific evaluation, and latent insight generation. We decompose the problem of learning the optimal KB by partitioning memory as local documents associated with semantic concepts, and solve the KB learning problem by our novel Expand-and-Gather Monte Carlo Tree Search (EG-MCTS) algorithm. Figure 1 provides an architecture overview of BREW, and Algorithm 1 in Appendix A describes the full pseudocode.

3.1 Trajectory Generation

Given the training dataset, we generate full-length trajectories, hereby referred to as rollouts, for each query using an LLM-powered agent conditioned on its associated KB. At initialization, the KB is empty, and we generate rollouts with an *empty* KB. Each rollout is evaluated using a correctness grader, which assigns a binary success label and an LLM based qualitative assessment against a set of human-validated general-purpose agent behavior rubrics (Biyani et al., 2024) (Step 1 in Figure 1). To enrich the diversity and coverage of these trajectories, we additionally apply hindsight relabeling (Section 3.2).

3.2 Hindsight Relabeling

To expand the diversity of training signal beyond the original rollouts, we apply hindsight relabeling (Andrychowicz et al., 2018). We sample failing trajectories and prompt an LLM to generate an alternative task description under which the agent’s observed behavior would constitute a valid solution. Re-executing the agent on this reframed task frequently succeeds, converting roughly half of the originally failing trajectories into passing ones and shifting the effective distribution.

3.3 Reflector and Integrator Agents

Reflector Agent: ReflAgent takes as input a rollout with its rubric and correctness labels, and outputs sentence-level insights with mapped concepts:

$$\{\text{concepts}, \text{insights}\} = \text{ReflAgent}(\{\text{rollout}, \text{eval}\}). \quad (1)$$

Examples of concept–insight pairs appear in Step 2 of Figure 1.

Concept Deduplication: Concept–insight pairs are annotated independently per rollout, often producing overlapping or paraphrased concepts. We address this via semantic clustering (Steps 3–4, Figure 1; Algorithm 1, line 3 in Appendix A): contextual embeddings for each concept are generated using an LLM, clustered, and each insight is mapped to its cluster representative. Details appear in Algorithms 2 and 3 in Appendix A.

Integrator Agent: IntegAgent incrementally builds and refines KB documents $\{d(s_i)\} \in \mathcal{D}(s_i)$ during environment interaction. Instead of a centralized memory, the KB is partitioned into local documents, each tied to a meta concept. This design enables (1) efficient, context-specific retrieval; (2) modular updates with minimal interference; and (3) natural alignment with task semantics, as deduplicated meta concepts capture meaningful behavioral abstractions. Unlike prior work assuming flat memory or dialogue histories, this structure is well-suited for long-horizon, procedural tasks where behaviors cluster around discrete skills.

The KB is dynamically populated: concepts central to the dataset receive more updates. At each state, for meta concept k , IntegAgent updates its document via $d_k(s_{i+1}) \leftarrow \text{IntegAgent}(k, \text{insights}_k, d_k(s_i))$, so the full KB is $\mathcal{D}(s_i) = \bigcup_{k \in \mathcal{K}} \{d_k(s_i)\}$. To reduce LLM variance we use Expand-and-Gather MCTS (EG-MCTS; Figure 2).

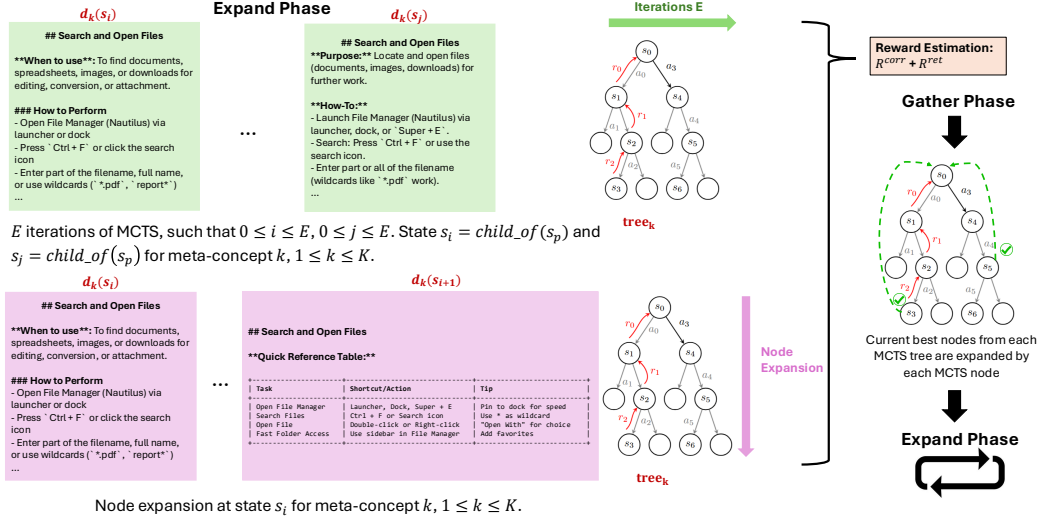


Figure 2: textmdIllustration of BREW’s KB optimization process using Expand-and-Gather MCTS with OSWORLD examples. In the **Expand Phase**, for each document k , we sample the best node from $tree_k$ using UCT and perform node expansion. Node rewards are estimated based on correctness and retrievability. In the **Gather Phase**, the best nodes from each tree are gathered at each node. The process is repeated for the next iteration of KB refinement.

3.4 Expand-and-Gather MCTS for Optimal KB Search

We initialize the meta-concept set $\mathcal{K}$ by deduplicating concepts extracted by ReflAgent from the first set of rollouts, then create an empty document per concept $k \in \mathcal{K}$.

We model finding $\mathcal{D}^*$ as a search over the state space of all possible KBs. By decomposing $\mathcal{D}$ into concept-level documents, the problem reduces to finding d_k^* for each concept k , with the full KB assembled as $\mathcal{D}^* = \bigcup_{k \in \mathcal{K}} \{d_k^*\}$. However, documents are *not* independent: an agent can retrieve any document during inference, making it difficult to assess changes in isolation. Expand-and-Gather MCTS (EG-MCTS) addresses this by searching the per-concept state spaces concurrently via parallel MCTS explorations synced after each iteration. Node expansions proceed independently, but reward calculation and insight generation are conditioned on a running-optimum KB state. Each iteration consists of two phases:

Expand Phase: For each search tree, we select the best state s^* and expand it concurrently. The expansion KB combines the current document $d_k(s^*)$ with the best documents for all other concepts: $\mathcal{D}_t = \{d_t\} \cup \{d_{i,i \neq t}^*\}$. Rollouts generated under $\mathcal{D}_t$ are consumed by ReflAgent for insights, after which IntegAgent produces updated document variants whose rewards are estimated and back-propagated through the tree.

Gather Phase: The current best states from each document’s MCTS tree are *gathered* and distributed to every tree, enabling (i) reward estimation for each expanded state and (ii) new insight generation for further expansion.

3.5 Reward-Guided Optimization

Reward Objective: Each document state is rewarded based on two complementary criteria: (i) how well the current document contributes to **accurate downstream reasoning**, and (ii) how **retrievable** it is in the context of a growing KB. Formally, the total reward at time step t is defined as:

$$R_t = \lambda_{\text{corr}} \cdot R_t^{\text{corr}} + \lambda_{\text{ret}} \cdot R_t^{\text{ret}} \quad (2)$$

where R_t^{corr} is the **correctness reward**, R_t^{ret} is the **retrieval reward**, and $\lambda_{\text{corr}}, \lambda_{\text{ret}} \in [0, 1]$ are scalar weights with $\lambda_{\text{corr}} + \lambda_{\text{ret}} = 1$.

Correctness Reward: The correctness reward R_t^{corr} evaluates the accuracy of the agent’s output over a held-out query set $\mathcal{Q}$, when reasoning over the current KB $\mathcal{D}_t$. It is defined as:

$$R^{\text{corr}}(d_t|\mathcal{D}_t) = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \text{Eval}_{\text{task}}(q, \text{agent} \oplus \mathcal{D}_t) \quad (3)$$

where $\text{Eval}_{\text{task}}$ is a task-specific evaluation function (e.g., question-answering accuracy, entailment correctness), and $\text{agent} \oplus \mathcal{D}_t$ denotes the agent acting over the hybrid KB.

Retrieval Reward: The retrieval reward R_t^{ret} measures how effectively the current document d_t can be retrieved from the current KB $\mathcal{D}_t$. For a held-out query set $\mathcal{Q}$, it is computed using the mean reciprocal rank (MRR):

$$R^{\text{ret}}(d_t|\mathcal{D}_t) = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \text{MRR}_q(d_t, \mathcal{D}_t) \quad (4)$$

3.6 Complexity Analysis

The overall time complexity of BREW is $O(|\mathcal{Q}_{\text{train}}| \cdot T_{\text{LLM}} + M \cdot |\mathcal{K}| \cdot h \cdot T_{\text{agent}})$, where $|\mathcal{Q}_{\text{train}}|$ is the number of training queries, T_{LLM} is the cost of a single LLM call, M is the number of MCTS iterations, $|\mathcal{K}|$ is the number of meta concepts, h is the number of candidates per expansion, and T_{agent} is the cost of a single agent rollout. See Algorithm 1 in Appendix A for the pseudocode and details about compute analysis in Appendix A.4.

4 Experimental Setup

Datasets We evaluate BREW on three benchmarks testing different interactive agent capabilities: OSWORLD for computer-use automation (Xie et al., 2024), τ^2 -Bench for tool use (Barres et al., 2025), and SPREADSHEETBENCH for data manipulation (Ma et al., 2024).

1. **OSWorld:** This benchmark tests multimodal agents on real-world computer tasks across 10 applications. We use *GTA1-7B*, a state-of-the-art computer-use agent with BREW. We split the data into 20 train and 349 test samples, balanced across application categories. Tasks are evaluated using 134 custom scripts that verify final application states.
2. **τ^2 -Bench:** This benchmark evaluates conversational agents on multi-turn tool-use scenarios across *Telecom*, *Retail*, and *Airline* domains. We test o4-mini-based tool-calling agent. We use 20 train and 254 test samples, balanced across domains.
3. **SpreadsheetBench:** This benchmark evaluates agents on real-world spreadsheet manipulation, spanning both cell-level and sheet-level tasks. It contains 912 authentic user instructions paired with 2,729 test cases (3 per instruction). We test o4-mini using a Python tool-calling agent, and enhance it with an embedding based Retrieval over the BREW KB generated over a small held-out train set of 30 samples.

Agents & KB Integration For OSWORLD, we augment *GTA1-7B* (Yang et al., 2025) (paired with O3 or GPT5.2 as grounding model). Retrieved documents are injected into the planner’s system prompt; for the grounding agent, retrieval is performed at each grounding step and provided in the user prompt. For τ^2 -Bench and SPREADSHEETBENCH, we augment tool-calling agents by prepending the top retrieved entries to the system prompt.

Baselines We compare BREW against two widely used experiential memory approaches, *Cognee*¹ (Markovic et al., 2025) and *Agent-Mem* (Xu et al., 2025c), both of which serve as established baselines for AI memory evaluation. *Cognee* uses a graph-plus-vector memory architecture to build cross-document connections from trajectories, while *Agent-Mem* provides a scalable memory layer for extracting and retrieving information from conversational data with graph-based representations.

¹github.com/topoteretes/cognee

Method	OSWorld Verified GTA1-7B + o3	OSWorld Verified GTA1-7B + GPT5.2	τ^2 Bench o4-mini	SpreadsheetBench o4-mini
Baseline	53.30	63.04	56.63	44.30
Cognee	46.70	58.55	57.71	42.10
Agent-Mem	43.83	57.28	52.69	42.00
BREW	61.70	68.34	62.50	48.50

Table 1: Comparison against baselines – task success rate, ratio of independent successes, and first-testcase pass rate for OSWorld Verified, τ^2 Bench, and SpreadsheetBench, respectively.

Other Experimental Configs: For all experiments, expansion width $e = 3$, max depth $k = 3$, and balanced reward weights $\lambda_{\text{corr}} = \lambda_{\text{ret}} = 0.5$. During MCTS node selection, we use the UCT (Kocsis and Szepesvári, 2006) for balancing exploration and exploitation. Full experimental details are provided in the Appendix.

5 Analysis & Discussion

In this section, we present findings from our evaluation of BREW. For more details on qualitative insights and discussion you may refer to the supplementary material.

5.1 Main Results

Table 1 summarizes performance across all three benchmarks. BREW outperforms every baseline on every benchmark, achieving absolute gains of +8.4 on OSWORLD (GTA1-7B+o3), +5.3 (GTA1-7B+GPT5.2), +5.87 on τ^2 -Bench, and +4.2 on SPREADSHEETBENCH. Notably, both Cognee and Agent-Mem sometimes *degrade* performance below the no-memory baseline (e.g., Agent-Mem on OSWORLD: 43.83 vs. 53.30; Cognee on SPREADSHEETBENCH: 42.10 vs. 44.30), suggesting that naïve memory injection can hurt when retrieved context is misaligned. BREW avoids this failure mode through reward-driven state search, which optimizes KB content before deployment rather than injecting raw trajectory fragments at inference time.

5.2 Variations Across State Search Strategy

BREW treats KB construction as a search problem over possible KB states. We compare MCTS against *Iterative Refinement* and *Greedy Search* (details in Appendix A.3). Table 3 presents results across all three benchmarks.

Iterative refinement is fragile under LLM stochasticity. It degrades below baseline on SPREADSHEETBENCH (42.98 vs. 44.30) and barely matches it on OSWORLD (53.10 vs. 53.30), with only a marginal gain on τ^2 -Bench (57.34 vs. 56.63) (Table 3). Without branching, a single bad refinement step has no recovery mechanism.

Greedy search suffices for simple domains; MCTS excels on complex ones. On OSWORLD, MCTS outperforms greedy by 3.7 points (56.80 vs. 53.10), while on τ^2 -Bench and SPREADSHEETBENCH the gap narrows (Table 3), consistent with OSWORLD’s richer inter-document dependencies benefiting from broader exploration.

5.3 BREW learns recipes from sub-trajectories in OSWORLD.

Figure 3 breaks down performance by OSWorld category. The OSWorld analysis identifies three principal failure modes for agents: GUI grounding, operational knowledge, and long-horizon planning. BREW’s KB directly addresses the second by distilling procedural recipes from trajectories.

Accuracy gains track operational-knowledge bottlenecks. Writer and Thunderbird, which rely on repetitive menu-driven workflows with reusable sub-procedures, show the strongest accuracy improvements. VSCode benefits similarly from distilled IDE command sequences. GIMP and Calc, by contrast, see negligible accuracy change: GIMP tasks are bottlenecked by fine-grained spatial manipulation, while Calc tasks demand per-instance formula logic with little cross-task procedural overlap.

Method	OSWorld Verified GTA1-7B + o3	OSWorld Verified GTA1-7B + GPT5.2	τ^2 Bench o4-mini	SpreadsheetBench o4-mini
BREW -Iterative	59.40	67.29	59.14	47.13
BREW -Greedy	59.40	67.29	60.71	46.66
BREW -MCTS	61.70	68.34	62.50	48.50

Table 2: Ablation over BREW variants. We compare search strategies (Iterative, Greedy, MCTS) and the effect of Hindsight Relabeling across all three benchmarks.

Efficiency decouples from correctness. The step-reduction signal (dashed line) reveals that even in categories where accuracy is flat, BREW reduces execution length by 14–23 steps. The KB is retrieved and followed for the operational-knowledge component, but the task ultimately fails at a downstream bottleneck the KB cannot address, such as a grounding misclick or a missing perceptual inference. This decoupling indicates that future progress in these categories requires complementing procedural memory with improved visual grounding rather than richer KBs.

5.4 BREW learns aggressive resolution strategies for τ^2 -Bench

Table 10, breaks down error types on τ^2 -Bench Retail. Prior failure analysis shows that *Wrong info* and *Wrong decision* errors are predominantly terminal: once the agent provides a miscalculated price or violates a policy constraint (e.g., calling exchange tools item-by-item when a batched call is required), the conversation diverges irrecoverably. BREW targets this by surfacing verified policy resolutions at retrieval time, preventing the initial misstep rather than attempting recovery. Due to this **BREW prevents terminal policy errors by front-loading verified resolutions at retrieval time**. Other baselines occasionally increase errors because τ -bench policies contain subtle conditional distinctions: return eligibility depends on order status, item category, and time window, and retrieving a precedent matching request type but differing on one condition injects a wrong constraint.

5.5 BREW learns domain-specific strategies for SpreadsheetBench

Domain-specific KB entries correct recurring baseline failure modes. BREW shows consistent improvements over the Baseline on SpreadsheetBench. We identified 109 cases where BREW succeeded on the first test case while the baseline failed, with identical inputs aside from KB augmentation. In most of these cases the difference traced to more precise formula placement, followed by result verification before submission and correct filter usage. Three failure modes account for the majority of corrected errors: (i) header misdetection, where the baseline promotes the first data row to column names in headerless spreadsheets; (ii) fragile formula construction, such as using string comparison instead of ISBLANK for blank-cell detection; and (iii) scope violations, where baseline agents write beyond the specified cell range or substitute Python lookups for requested Excel formulas.

KB-guided validation prevents cascading errors. The Header Extraction entry guides the agent to validate column metadata before processing, while the Writeback Results and Column Selection entries enforce range compliance and appropriate formula idioms. Beyond correctness, BREW-augmented trajectories consistently include verification steps such as diff generation, output reloading, and formula spot-checks, surfaced through the Difference in State KB entry. These routines catch errors early in multi-step manipulation chains that would otherwise propagate silently. We provide full case studies in Appendix A.7.3.

5.6 Effect of Hindsight Relabeling

Hindsight relabeling consistently improves task success rates across all benchmarks and search strategies (Table 3). The key insight is that the same agent behavior receives opposite polarity from the Reflector depending on whether the trajectory is labeled as passing or failing (Eq. 1). A failing trajectory produces diagnostic insights (“what went wrong”), while

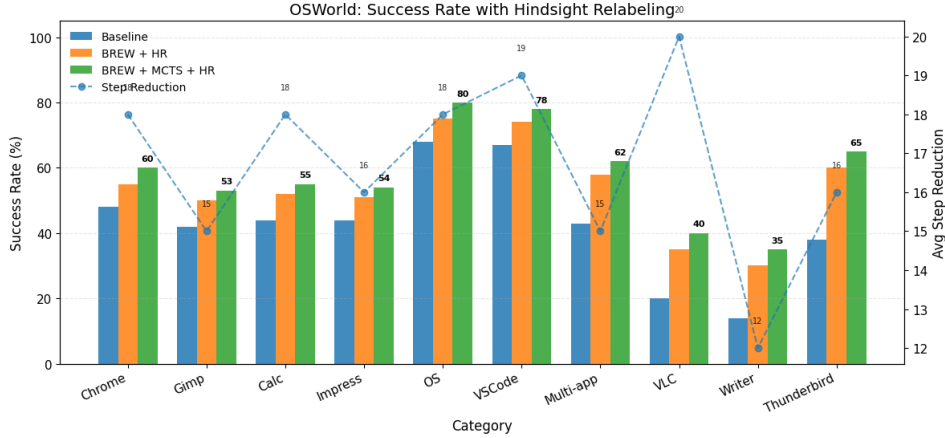


Figure 3: The bar plot represents the category-wise success rate over various tasks in the OSWORLD dataset over the GTA1-agent, whereas the line plot demonstrates the reduction in the number of steps for the successful cases. Note that even in scenarios where the KB doesn’t help increase the success rate, it steeply reduces the #steps needed to succeed.

the same behavioral pattern, relabeled as passing under a new task, produces prescriptive insights (“what to do”). This dual signal enriches the KB with both cautionary and actionable procedural knowledge. Additionally, re-execution on the relabeled task is not guaranteed to reproduce the original trajectory exactly, exposing nearby behavioral modes that give the Reflector genuinely novel material.

The resulting shift from 50/50 to roughly 66% passing / 33% failing biases KB construction toward diverse successful demonstrations, representing latent competences never surfaced as successes in the original data. While retaining enough failure coverage to prevent overconfidence. See Appendix A.2 for the full analysis.

5.7 Knowledge Base Fidelity on Training Trajectories

Table 8 reports train-set accuracy as a proxy for how faithfully each method’s KB encodes knowledge from its construction trajectories. All memory-augmented methods outperform the memoryless baseline on the train split, but cross-referencing with test results (Table 1) reveals that high train fidelity can be actively harmful. Agent-Mem matches BREW on τ^2 -Bench train yet collapses to 52.69 on test, below the memoryless baseline. Cognee shows the same inversion on OSWorld. We find Agent-Mem stores near-verbatim episodic traces, so retrieval at test time surfaces action sequences tightly coupled to the original task’s state transitions; minor shifts in goal or initial state turn the retrieved trace into a misleading prior. Cognee’s entity-centric graph retains relational structure but loses procedural ordering, reconstructing what entities were involved but not *in what sequence* or *under what conditions*. Both pathologies yield negative transfer: the KB retrieves confidently but incorrectly. BREW avoids this because the Reflector distills trajectories into condition-action rules indexed by environment state rather than by source task identity. Retrieval therefore generalizes across tasks sharing procedural sub-structure even when surface-level goals differ, making BREW the only method where train fidelity consistently converts to test gains across all three benchmarks.

6 Conclusions

BREW constructs and refines an interpretable, structured knowledge base from past agent interactions, offering an alternative to direct model fine-tuning. By organizing memory into concept-level documents and refining them via state-space search, BREW improves task success and efficiency across OSWORLD, τ^2 -Bench, and SPREADSHEETBENCH at manageable computational cost. Its effectiveness remains sensitive to data quality and coverage, motivating future work on adaptive refinement and tighter integration with agent planning.

References

- Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnab Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, July 2025.
- Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2018. URL <https://arxiv.org/abs/1707.01495>.
- Anthropic. Introducing computer use, a new Claude 3.5 Sonnet, and Claude 3.5 Haiku, October 2024. URL <https://www.anthropic.com/news/3-5-models-and-computer-use>. Accessed: 2025.
- Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2 -bench: Evaluating conversational agents in a dual-control environment, 2025. URL <https://arxiv.org/abs/2506.07982>.
- Param Biyani, Yasharth Bajpai, Arjun Radhakrishna, Gustavo Soares, and Sumit Gulwani. Rubicon: Rubric-based evaluation of domain-specific human ai conversations. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 161–169, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706851. doi: 10.1145/3664646.3664778. URL <https://doi.org/10.1145/3664646.3664778>.
- Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.
- Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B. Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning, 2020. URL <https://arxiv.org/abs/2006.08381>.
- Naman Gupta, Shashank Kirtania, Priyanshu Gupta, Krishna Kariya, Sumit Gulwani, Arun Iyer, Suresh Parthasarathy, Arjun Radhakrishna, Sriram K. Rajamani, and Gustavo Soares. Stackfeed: Structured textual actor-critic knowledge base editing with feedback, 2024a. URL <https://arxiv.org/abs/2410.10584>.
- Priyanshu Gupta, Shashank Kirtania, Ananya Singha, Sumit Gulwani, Arjun Radhakrishna, Gustavo Soares, and Sherry Shi. MetaReflection: Learning instructions for language agents using past reflections. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 8369–8385, Miami, Florida, USA, November 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.477. URL <https://aclanthology.org/2024.emnlp-main.477/>.
- Maryam Hashemzadeh, Elias Stengel-Eskin, Sarath Chandar, and Marc-Alexandre Cote. Sub-goal distillation: A method to improve small language agents, 2024. URL <https://arxiv.org/abs/2405.02749>.
- Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating memory in llm agents via incremental multi-turn interactions, 2025. URL <https://arxiv.org/abs/2507.05257>.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL <https://openreview.net/forum?id=VTF8yNQm66>.
- Zaid Khan, Elias Stengel-Eskin, Archiki Prasad, Jaemin Cho, and Mohit Bansal. Executable functional abstractions: Inferring generative programs for advanced math problems. 2025.

- Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46056-5.
- Xinzhe Li. A review of prominent paradigms for llm-based agents: Tool use, planning (including rag), and feedback learning. In *Proceedings of the 31st International Conference on Computational Linguistics (COLING)*, pages 9760–9779, Abu Dhabi, UAE, 2025. Association for Computational Linguistics. URL <https://aclanthology.org/2025.coling-main.652>.
- Michael L. Littman. An optimization-based categorization of reinforcement learning environments. 1993. URL <https://api.semanticscholar.org/CorpusID:17988064>.
- Yixiu Liu, Yang Nan, Weixian Xu, Xiangkun Hu, Lyumanshan Ye, Zhen Qin, and Pengfei Liu. Alphago moment for model architecture discovery. *ArXiv*, abs/2507.18074, 2025. URL <https://api.semanticscholar.org/CorpusID:280018530>.
- Zeyao Ma, Bohan Zhang, Jing Zhang, Jifan Yu, Xiaokang Zhang, Xiaohan Zhang, Sijia Luo, Xi Wang, and Jie Tang. Spreadsheetbench: Towards challenging real world spreadsheet manipulation. *Advances in Neural Information Processing Systems*, 37:94871–94908, 2024.
- Vasilije Markovic, Lazar Obradovic, Laszlo Hajdu, and Jovan Pavlovic. Optimizing the interface between knowledge graphs and llms for complex reasoning, 2025. URL <https://arxiv.org/abs/2505.24478>.
- Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav M. Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, Matej Balog, and Google Deepmind. Alphaevolve: A coding agent for scientific and algorithmic discovery. *ArXiv*, abs/2506.13131, 2025. URL <https://api.semanticscholar.org/CorpusID:278658695>.
- OpenAI. Introducing Operator, January 2025. URL <https://openai.com/index/introducing-operator/>. Accessed: 2025.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL <https://arxiv.org/abs/2310.08560>.
- Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326*, 2025.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL <https://arxiv.org/abs/2305.18290>.
- John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. In *Proceedings of the 34th International Conference on Machine Learning (ICML 2017)*, 2017. URL <https://arxiv.org/abs/1707.06347>.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL <https://arxiv.org/abs/2402.03300>.
- David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL <http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html>.

- Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*, 2023.
- Yu Wang, Chi Han, Tongtong Wu, Xiaoxin He, Wangchunshu Zhou, Nafis Sadeq, Xiusi Chen, Zexue He, Wei Wang, Gholamreza Haffari, Heng Ji, and Julian McAuley. Towards lifespan cognitive systems, 2025. URL <https://arxiv.org/abs/2409.13265>.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Oworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 52040–52094. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5d413e48f84dc61244b6be550f1cd8f5-Paper-Datasets_and_Benchmarks_Track.pdf.
- Ran Xu, Yuchen Zhuang, Yue Yu, Haoyu Wang, Wenqi Shi, and Carl Yang. Rag in the wild: On the (in)effectiveness of llms with mixture-of-knowledge retrieval augmentation. *arXiv preprint arXiv:2507.20059*, 2025a.
- Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*, 2025b.
- Wujiang Xu, Kai Mei, Hang Gao, Juntao Tan, Zujie Liang, and Yongfeng Zhang. A-mem: Agentic memory for llm agents, 2025c. URL <https://arxiv.org/abs/2502.12110>.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.
- Yan Yang, Dongxu Li, Yutong Dai, Yuhao Yang, Ziyang Luo, Zirui Zhao, Zhiyuan Hu, Junzhe Huang, Amrita Saha, Zeyuan Chen, Ran Xu, Liyuan Pan, Caiming Xiong, and Junnan Li. Gta1: Gui test-time scaling agent, 2025. URL <https://arxiv.org/abs/2507.05791>.

A Appendix

A.1 Details of the BREW Algorithm

We provide pseudocode for the core components of BREW, aligning with the stages introduced in Section 3. Algorithm 1 describes the overall BREW procedure. GENERATEINSIGHTS (Alg. 2) produces concept-aligned insights from annotated rollouts using ReflAgent. DEDUPLICATECONCEPTS (Alg. 3) clusters semantically overlapping concepts into a compact meta-concept set. INTEGAGENT incrementally builds and updates per-concept documents using newly generated insights. Finally, EXPANDNODE (Alg. 4) performs MCTS-guided expansions to explore improved document variants, while EVALUATE (Alg. 5) scores candidate KB states using correctness and retrieval-based rewards.

Algorithm 1 BREW: Bootstrapping Experientially-learned Environmental Knowledge

Require: Training samples $\mathcal{Q}_{\text{train}}$, eval samples $\mathcal{Q}_{\text{eval}}$, rubrics, iterations M , candidates per expansion h

Ensure: Optimized KB $\mathcal{D}^*$

Initialization

```

1:  $\mathcal{D}_0 \leftarrow \emptyset$ 
2:  $\mathcal{B} \leftarrow \text{GENERATEINSIGHTS}(\mathcal{Q}_{\text{train}}, \mathcal{D}_0, \text{rubrics})$ 
3:  $\mathcal{B} \leftarrow \mathcal{B} \cup \text{HINDSIGHTRELABEL}(\mathcal{Q}_{\text{train}}, \mathcal{D}_0, \text{rubrics})$  ▷ Relabel failing trajectories
4:  $\mathcal{K} \leftarrow \text{DEDUPLICATECONCEPTS}(\mathcal{B})$  ▷ Initial concept set
5: for each  $k \in \mathcal{K}$  do
6:    $d_k^0 \leftarrow \text{INTEGAGENT}(k, \mathcal{I}_k, \emptyset)$ 
7:   Initialize  $\text{tree}_k$  with root node  $d_k^0$ 
8: end for
9:  $\mathcal{D}_{\text{current}} \leftarrow \bigcup_{k \in \mathcal{K}} \{d_k^0\}$  ▷ Initial KB

```

EG-MCTS Optimization

```

10: for  $t = 1$  to  $M$  do ▷ Parallel expansion across concepts
11:   for each  $k \in \mathcal{K}$  do
12:      $s_k \leftarrow \text{SELECTBESTNODE}(\text{tree}_k)$  ▷ UCT selection
13:      $\mathcal{D}_{\text{best}} \leftarrow \bigcup_{k' \in \mathcal{K}} \{d_{k'}^{\text{best}}\}$  ▷ Current best docs
14:      $\text{EXPANDNODE}(s_k, k, h, \mathcal{D}_{\text{current}}, \mathcal{D}_{\text{best}}, \text{tree}_k)$ 
15:   end for ▷ Update current best documents
16:   for each  $k \in \mathcal{K}$  do
17:      $d_k^{\text{best}} \leftarrow \text{best document in tree}_k$ 
18:   end for
19:    $\mathcal{D}_{\text{current}} \leftarrow \bigcup_{k \in \mathcal{K}} \{d_k^{\text{best}}\}$ 
20: end for
21: return  $\mathcal{D}_{\text{current}}$ 

```

Time Complexity: $O(|\mathcal{Q}_{\text{train}}| \cdot T_{\text{LLM}} + M \cdot |\mathcal{K}| \cdot h \cdot T_{\text{agent}})$

Algorithm 2 GenerateInsights: Extract behavioral insights from trajectories

Require: Queries $\mathcal{Q}$, KB $\mathcal{D}$, rubrics

Ensure: Concept-insight pairs $\mathcal{B}$

```

1:  $\mathcal{B} \leftarrow \emptyset$ 
2: for each query  $q \in \mathcal{Q}$  do
3:    $\tau \leftarrow \text{LLM}(q, \mathcal{D})$  ▷ Generate trajectory
4:    $\text{label} \leftarrow \text{GRADE}(\tau)$  ▷ Success/failure
5:    $(c, i) \leftarrow \text{REFLAGENT}(\tau, \text{rubrics}, \text{label})$ 
6:    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(c, i, q)\}$  ▷ Store with source query
7: end for
8: return  $\mathcal{B}$ 

```

Algorithm 3 DeduplicateConcepts: Cluster similar concepts and map queries**Require:** Concept-insight-query triples $\mathcal{B}$ **Ensure:** Meta-concepts $\mathcal{K}$ with mapped queries and insights

- 1: Extract all concepts from $\mathcal{B}$
- 2: Embed and cluster concepts by similarity
- 3: $\mathcal{K} \leftarrow$ cluster representatives
- 4: **for** each $k \in \mathcal{K}$ **do**
- 5: $\mathcal{Q}_k^{\text{train}} \leftarrow$ {training queries that contributed insights to k }
- 6: $\mathcal{Q}_k^{\text{eval}} \leftarrow$ {held-out queries relevant to k }
- 7: $\mathcal{I}_k \leftarrow$ {all insights mapped to concept k }
- 8: **end for**
- 9: **return** $\mathcal{K}$ with associated queries and insights

Algorithm 4 ExpandNode: Generate and evaluate new document variants**Require:** Node s , concept k , candidates h , current KB $\mathcal{D}_{\text{current}}$, best docs $\mathcal{D}_{\text{best}}$, tree**Ensure:** Updated tree with new evaluated nodes

- 1: ▷ Generate new insights from concept-relevant queries
- 2: $\mathcal{B}_{\text{new}} \leftarrow \emptyset$
- 3: **for** query $q \in \mathcal{Q}_k^{\text{train}}$ **do**
- 4: $\tau \leftarrow \text{LLM}(q, \mathcal{D}_{\text{current}})$
- 5: $(c, i) \leftarrow \text{ANNOTATE}(\tau, \text{rubrics}, \cdot)$
- 6: **if** c maps to k **then**
- 7: $\mathcal{B}_{\text{new}} \leftarrow \mathcal{B}_{\text{new}} \cup \{i\}$
- 8: **end if**
- 9: **end for**
- 10: ▷ Generate and evaluate candidate documents
- 11: **for** $j = 1$ to h **do**
- 12: $d_{k,j} \leftarrow \text{INTEGAGENT}(k, \mathcal{I}_k \cup \mathcal{B}_{\text{new}}, d_k^s)$
- 13: ▷ Evaluate using hybrid KB with best docs from other concepts
- 14: $\mathcal{D}_{\text{hybrid}} \leftarrow \{d_{k,j}\} \cup \{d_{k'} \in \mathcal{D}_{\text{best}} : k' \neq k\}$
- 15: $R_{k,j} \leftarrow \text{EVALUATE}(d_{k,j}, \mathcal{D}_{\text{hybrid}}, \mathcal{Q}_k^{\text{eval}})$
- 16: ▷ Add to tree and backpropagate
- 17: Add $(d_{k,j}, R_{k,j})$ as child of s in tree
- 18: Backpropagate $R_{k,j}$ from new node to root
- 19: **end for**

Algorithm 5 Evaluate: Score document using held-out queries**Require:** Document d_k , hybrid KB $\mathcal{D}_{\text{hybrid}}$, eval queries $\mathcal{Q}_k^{\text{eval}}$ **Ensure:** Reward score R

- 1: $R^{\text{corr}} \leftarrow 0$
- 2: $R^{\text{ret}} \leftarrow 0$
- 3: **for** each $q \in \mathcal{Q}_k^{\text{eval}}$ **do**
- 4: $R^{\text{corr}} \leftarrow R^{\text{corr}} + \text{EVAL}(q, \text{agent} \oplus \mathcal{D}_{\text{hybrid}})$
- 5: $R^{\text{ret}} \leftarrow R^{\text{ret}} + \text{MRR}(d_k, q, \mathcal{D}_{\text{hybrid}})$
- 6: **end for**
- 7: $R^{\text{corr}} \leftarrow \frac{R^{\text{corr}}}{|\mathcal{Q}_k^{\text{eval}}|}$
- 8: $R^{\text{ret}} \leftarrow \frac{R^{\text{ret}}}{|\mathcal{Q}_k^{\text{eval}}|}$
- 9: **return** $\lambda_{\text{corr}} \cdot R^{\text{corr}} + \lambda_{\text{ret}} \cdot R^{\text{ret}}$

A.1.1 Integrator Agent Prompt

We specify the IntegAgent prompt below:

Enhanced Documentation Editor Prompt

You are a meticulous documentation-level editor specializing in comprehensive technical reference materials. You will be given a list of topic nodes, each containing structured information that must be preserved and enhanced with maximum detail retention.

Input Structure Analysis

Each node contains:

- **Title**: The primary topic identifier
- **Context**: Background information and conceptual foundation
- **How to Use**: Step-by-step instructions, commands, flags, parameters, and implementation details
- **When to Use**: Specific scenarios, conditions, and decision criteria
- **Best Practices**: Expert recommendations, optimization techniques, and common pitfalls to avoid

Detailed Processing Requirements

1. Information Preservation (Zero Loss Policy)

- **Preserve every technical detail**: All command-line flags, parameter values, configuration options, file paths, URLs, version numbers, and exact syntax
- **Maintain all examples**: Keep every code snippet, sample input/output, file names, directory structures, and command sequences exactly as provided
- **Retain contextual nuances**: Preserve qualifying language like ``typically,`` ``usually,`` ``in most cases,`` ``when available,`` and conditional statements
- **Keep quantitative data**: Preserve all numbers, measurements, timeframes, limits, thresholds, and statistical information
- **Maintain cross-references**: Keep all mentions of related tools, dependencies, prerequisites, and interconnected concepts

2. Enhanced Detail Extraction

- **Expand abbreviations**: When encountering shortened forms, expand them naturally while preserving the original
- **Surface implicit knowledge**: Make obvious assumptions explicit (e.g., "this requires root permissions," "assumes default configuration")
- **Clarify relationships**: Explicitly describe how different components, options, or steps relate to each other
- **Highlight edge cases**: Emphasize special conditions, exceptions, or unusual scenarios mentioned in the source
- **Elaborate on consequences**: When the source mentions outcomes, expand on both success and failure scenarios

3. Prose Transformation Guidelines

- **Bullet integration**: Transform each bullet point into 1-3 complete sentences that naturally flow together
- **Technical precision**: Use precise technical vocabulary while maintaining readability
- **Logical flow**: Organize information within each section to follow a logical sequence (setup -> execution -> verification)
- **Contextual embedding**: Weave code snippets and technical terms seamlessly into narrative sentences
- **Comprehensive coverage**: Ensure every sub-bullet, nested item, and parenthetical note becomes part of the prose

4. Structural Requirements

- **Heading hierarchy**: Use `# Title` for each node's main heading
- **Section order**: Maintain Context -> How to Use -> When to Use -> Best Practices sequence

- **Paragraph organization**: Create substantial paragraphs (3-6 sentences) rather than brief statements
- **Transition quality**: Craft smooth bridges between sections and between different nodes
- **Code formatting**: Preserve all inline code with backticks and maintain proper formatting for code blocks

5. Quality Assurance Checklist

Before finalizing, verify:

- [] Every piece of source information appears in the output
- [] All technical specifications, parameters, and examples are intact
- [] Code snippets maintain their exact syntax and formatting
- [] Prose flows naturally without choppy or fragmented sentences
- [] Each section provides comprehensive coverage of its topic area
- [] Cross-references and dependencies are clearly explained
- [] No section labels or formatting artifacts remain in the prose

Output Specifications

Generate a single, cohesive markdown document that reads as authoritative technical documentation. The result should be comprehensive enough that a reader could successfully implement the described tools or techniques using only the information provided, without referring back to the original nodes.

Input Nodes:

```
<NODES>
{node_list}
</NODES>
```

Now, produce the aggregated markdown reference sheet with maximum detail preservation and enhanced clarity.

A.2 Hindsight Relabeling: Details

Algorithm 6 formalizes the hindsight relabeling procedure introduced in Section 3.2.

Algorithm 6 HindsightRelabel: Augment trajectory pool via task reframing

Require: Queries $\mathcal{Q}$, KB $\mathcal{D}$, rubrics

Ensure: Augmented concept-insight pairs $\mathcal{B}_{\text{HR}}$

```
1:  $\mathcal{B}_{\text{HR}} \leftarrow \emptyset$ 
2:  $\mathcal{T}_{\text{pass}}, \mathcal{T}_{\text{fail}} \leftarrow \text{SPLITBYOUTCOME}(\mathcal{Q}, \mathcal{D})$ 
3:  $\mathcal{T}_{\text{sample}} \leftarrow \text{BALANCEDSAMPLE}(\mathcal{T}_{\text{pass}}, \mathcal{T}_{\text{fail}}, 0.5)$ 
4: for each failing trajectory  $\tau_f \in \mathcal{T}_{\text{sample}}$  do
5:   summary  $\leftarrow \text{SUMMARIZE}(\tau_f)$  ▷ Key decisions, deviation point
6:    $q' \leftarrow \text{LLM}(\text{summary})$  ▷ Generate reframed task
7:    $\tau' \leftarrow \text{EXECUTE}(q', \mathcal{D})$  ▷ Re-run agent on new task
8:   label'  $\leftarrow \text{GRADE}(\tau')$ 
9:    $(c, i) \leftarrow \text{REFLAGENT}(\tau', \text{rubrics}, \text{label}')$ 
10:   $\mathcal{B}_{\text{HR}} \leftarrow \mathcal{B}_{\text{HR}} \cup \{(c, i, q')\}$ 
11: end for
12: return  $\mathcal{B}_{\text{HR}}$ 
```

Mechanism. We sample an equal number of passing and failing trajectories from the training set. For each failing trajectory, we construct a concise summary capturing the agent’s key decisions, intermediate states, and the point of deviation from correct behavior. An LLM then generates a new task description that is *consistent with the agent’s observed*

Method	OSWorld Verified GTA1-7B + o3	OSWorld Verified GTA1-7B + GPT5.2	τ^2 Bench o4-mini	SpreadsheetBench o4-mini
BREW -Iterative	53.10	62.71	57.34	42.98
BREW -Iterative—HR	59.40	67.29	59.14	47.13
BREW -Greedy	53.10	62.71	59.14	45.94
BREW -Greedy—HR	59.40	67.29	60.71	46.66
BREW -MCTS	56.80	65.83	59.14	46.80
BREW -MCTS—HR	61.70	68.34	62.50	48.50

Table 3: Ablation over BREW variants. We compare search strategies (Iterative, Greedy, MCTS) and the effect of Hindsight Relabeling across all three benchmarks.

behavior rather than the original task. By conditioning task generation on the trajectory, the reframed task remains grounded in realistic execution patterns rather than arbitrary synthetic data.

The agent is re-executed on the generated task. Because the new task was designed to align with what the agent naturally did, re-execution frequently succeeds. Empirically, approximately half of the originally failing trajectories become passing after relabeling, shifting the effective data distribution from a 50/50 pass/fail split to roughly 66% passing and 33% failing.

Why relabeled trajectories carry different information. The Reflector agent takes as input the trajectory *together with* its correctness label and rubric evaluation. The polarity of this label fundamentally changes the nature of every extracted insight. When the Reflector encounters a failing trajectory, insights are diagnostic: “this approach does not work.” When the same behavioral pattern appears in a relabeled trajectory that passes, insights become prescriptive: “when facing this kind of task, performing X leads to success.” Without relabeling, competent-but-misaligned behaviors only ever contribute negative signal, and the procedural knowledge they embody is never surfaced as reusable recipes.

A second effect compounds this: re-execution on the reframed task is not guaranteed to reproduce the original trajectory exactly. The agent may follow nearby but distinct action sequences, exploring the local neighborhood of its decision boundary and exposing additional behavioral modes absent from the original dataset.

Effect on KB quality. The shift to $\sim 66/33$ passing/failing has two consequences. First, the KB receives a larger pool of *diverse* successful demonstrations—not redundant with the original passing set, but representing behavioral modes near the agent’s decision boundary. Second, the remaining failing trajectories still provide negative signal, preventing brittle overconfidence. The resulting asymmetry biases KB construction toward actionable, prescriptive knowledge while preserving failure coverage.

A.3 Search Strategy Details

In addition to EG-MCTS (Section 3.4), we evaluate two simpler baselines for KB state search:

Iterative Refinement. For each meta-concept k , we initialize an empty document d_k and repeatedly invoke the Integrator agent to refine it using newly generated insights. At each step the current document is replaced by the Integrator’s output, regardless of quality. This process repeats for up to B iterations (we use $B = E \times K$ to match the total budget of MCTS with expansion width E and depth K). Because there is only a single trajectory through document space—no branching, no comparison—any degradation introduced by a single LLM call is permanently committed.

Greedy Search. At each expansion step for concept k , we generate E candidate documents in parallel (matching MCTS width) and select the one with the highest reward. The selected candidate replaces the current document, and the process repeats up to depth K . Unlike

Iterative Refinement, Greedy Search compares alternatives at each level, providing some robustness to LLM stochasticity. However, it commits entirely to the locally best candidate and never revisits rejected branches, making it susceptible to local optima in complex KB landscapes.

A.4 Computational Cost and LLM Call Analysis

We provide a detailed breakdown of the computational resources required by BREW and BREW-HR (with Hindsight Relabeling), addressing the cost of KB construction, the MCTS optimization overhead, the additional cost introduced by hindsight relabeling, and the amortized inference-time cost. All experiments were conducted on a single NVIDIA A100 GPU with 4 parallel processes for node expansion.

A.4.1 Time Complexity

BREW’s overall time complexity is:

$$O(|\mathcal{Q}_{\text{train}}| \cdot T_{\text{LLM}} + |\mathcal{Q}_{\text{train}}| \cdot T_{\text{agent}} + M \cdot |\mathcal{K}| \cdot h \cdot T_{\text{agent}} + M \cdot |\mathcal{K}| \cdot h \cdot T_{\text{eval}}),$$

where T_{LLM} and T_{agent} denote the wall-clock time for a single LLM call and a full agent episode respectively, M is the number of EG-MCTS iterations, h is the number of candidate expansions per step, $|\mathcal{K}|$ is the number of meta-concepts, and T_{eval} is the time for one evaluation episode.

In practice, $T_{\text{LLM}} \ll T_{\text{agent}}$ and $T_{\text{eval}} \ll T_{\text{agent}}$, so the perceived complexity reduces to:

$$O((|\mathcal{Q}_{\text{train}}| + M \cdot |\mathcal{K}| \cdot h) \cdot T_{\text{agent}}).$$

Since each GENERATEINSIGHTS step produces at most l concepts per trajectory, we have $|\mathcal{K}| < l \cdot |\mathcal{Q}_{\text{train}}|$, yielding a final complexity of:

$$O(M \cdot l \cdot h \cdot |\mathcal{Q}_{\text{train}}| \cdot T_{\text{agent}}).$$

Empirically, $l \approx 1\text{--}2$ across all benchmarks, confirming that concept growth is sublinear relative to training set size.

Additional cost of Hindsight Relabeling. When Hindsight Relabeling is enabled (BREW-HR), the initialization phase incurs additional cost from three sources: (i) summarizing each failing trajectory, (ii) generating a reframed task description via an LLM call, and (iii) re-executing the agent on the reframed task. For a balanced sample of $|\mathcal{T}_{\text{fail}}|$ failing trajectories (Algorithm 6), this adds:

$$O(|\mathcal{T}_{\text{fail}}| \cdot (T_{\text{LLM}} + T_{\text{agent}}))$$

to the initialization cost. Since we sample an equal number of passing and failing trajectories, $|\mathcal{T}_{\text{fail}}| \approx \frac{1}{2}|\mathcal{Q}_{\text{train}}|$. The re-execution step dominates ($T_{\text{LLM}} \ll T_{\text{agent}}$), so the effective overhead is approximately $\frac{1}{2}|\mathcal{Q}_{\text{train}}| \cdot T_{\text{agent}}$ additional agent runs during initialization.

Importantly, the MCTS optimization phase itself is not fundamentally more expensive under BREW-HR: although the initial insight pool $\mathcal{B}$ is enriched (containing both original and relabeled insights), the number of meta-concepts $|\mathcal{K}|$ after deduplication increases only modestly (see Table 4), and the MCTS search parameters (M, h, k) remain unchanged. The primary cost increase is therefore concentrated in the one-time initialization phase.

A.4.2 Meta-Concept Statistics

Table 4 reports the number of meta-concepts discovered per benchmark after deduplication, for both BREW and BREW-HR. Even for complex domains such as OSWORLD, which spans 10 application categories and 369 tasks, only 23–27 meta-concepts are required. The modest increase under BREW-HR reflects the fact that hindsight relabeling surfaces prescriptive knowledge from behavioral modes already present in the training data, which largely map onto existing concept clusters rather than creating entirely new ones.

This consolidation is driven by the semantic clustering in DEDUPLICATECONCEPTS (Algorithm 3), which achieves a ratio of approximately 1.15 concepts per training trajectory on OSWORLD for BREW and 1.35 for BREW-HR.

Benchmark	Training Samples	Meta-Concepts ($ \mathcal{K} $) BREW	BREW-HR	Concepts / Trajectory
OSWORLD	20	23	27	1.15 / 1.35
τ^2 -Bench	20	8	10	0.40 / 0.50
SPREADSHEETBENCH	30	26	30	0.87 / 1.00

Table 4: Number of meta-concepts after deduplication for BREW and BREW-HR. Hindsight relabeling introduces a modest increase in concepts (15–25%) because relabeled trajectories largely consolidate into existing semantic clusters.

A.4.3 LLM Call Breakdown

Table 5 provides a per-benchmark breakdown of LLM calls during KB construction for both BREW and BREW-HR, alongside agent evaluation costs. The KB construction calls represent a *one-time offline cost* that is amortized across all test episodes.

Benchmark	KB Construction LLM Calls BREW	BREW-HR	HR Overhead	Calls / Agent Run (Eval)	Test Episodes	Amortized Cost / Episode BREW	BREW-HR
OSWORLD	1,192	1,680	+41%	~21,000	369	~3.2	~4.6
τ^2 -Bench	1,223	1,715	+40%	~22,000	254	~4.8	~6.8
SPREADSHEETBENCH	1,397	1,955	+40%	~5,400	2,729	~0.5	~0.7

Table 5: LLM call statistics for BREW and BREW-HR. KB construction calls are a one-time offline cost. Hindsight relabeling adds ~40% more construction calls due to trajectory summarization, task reframing, and agent re-execution. The amortized per-episode overhead remains negligible relative to evaluation-time calls.

The ~40% increase in construction calls under BREW-HR arises from three sources: (i) LLM calls for summarizing $\sim \frac{1}{2} |Q_{\text{train}}|$ failing trajectories, (ii) LLM calls for generating reframed task descriptions, and (iii) agent re-execution on the reframed tasks, which itself involves multiple LLM calls per episode. The cost increase is proportional to the number of failing trajectories in the balanced sample. Even with this overhead, the amortized per-episode cost remains small: for OSWORLD, BREW-HR adds only ~4.6 calls per test episode compared to ~21,000 calls during evaluation, a ratio of 0.02%.

Why the overhead is justified. The key insight behind Hindsight Relabeling is that the same agent behavior receives opposite polarity from the Reflector depending on whether the trajectory is labeled as passing or failing. A failing trajectory produces diagnostic insights (“what went wrong”), while the same behavioral pattern, relabeled as passing under a reframed task, produces prescriptive insights (“what to do”). This dual signal enriches the KB with both cautionary and actionable procedural knowledge. Additionally, re-execution on the relabeled task is not guaranteed to reproduce the original trajectory exactly, exposing nearby behavioral modes that give the Reflector genuinely novel material.

The resulting shift from a 50/50 pass/fail distribution to roughly 66% passing and 33% failing biases KB construction toward diverse successful demonstrations, representing latent competences never surfaced as successes in the original data, while retaining enough failure coverage to prevent overconfidence. As shown in Table 8, BREW-HR achieves the highest train-set accuracy on OSWORLD (84.00% vs. 78.00% for BREW) and competitive performance on other benchmarks, confirming that the additional construction cost translates into measurably better KB quality.

A.4.4 MCTS Optimization Overhead

Table 6 reports wall-clock times for KB construction under the default MCTS configuration ($M=10$, $h=3$, $k=3$) for both BREW and BREW-HR. We additionally compare against greedy construction to isolate the overhead introduced by MCTS exploration and hindsight relabeling.

Benchmark	Greedy	BREW (MCTS)	BREW-HR (MCTS)	HR Overhead vs. BREW
OSWORLD	~1h 10min	~1h 25min	~1h 55min	+35%
τ^2 -Bench	~1h 00min	~1h 15min	~1h 45min	+40%
SPREADSHEETBENCH	~1h 15min	~1h 30min	~2h 05min	+39%

Table 6: Wall-clock time for KB construction (NVIDIA A100, 4 parallel processes). MCTS adds ~15–20% overhead relative to greedy construction. Hindsight relabeling adds a further ~35–40% over BREW due to the additional trajectory summarization, reframing, and re-execution during initialization.

The MCTS optimization adds approximately 15–20% wall-clock time compared to greedy construction, while Hindsight Relabeling adds a further ~35–40% on top of BREW. The combined overhead of BREW-HR over greedy construction ranges from ~55–65%. This overhead is concentrated entirely in the offline initialization phase and is justified by consistent performance gains: for example, BREW-HR achieves the strongest results on OSWORLD (Table ??), and the additional cost is amortized across hundreds to thousands of test episodes.

A.4.5 End-to-End Cost Breakdown for OSWORLD

To provide a concrete picture, we decompose the full pipeline for both BREW and BREW-HR on OSWORLD:

Phase	Computation	BREW	BREW-HR
Trajectory generation	$20 \times \sim 2.5$ min/trajectory	~50 min	~50 min
Insight extraction & dedup	GENERATEINSIGHTS + DEDUPLICATECONCEPTS	~10 min	~10 min
Hindsight relabeling	Summarize + reframe + re-execute (~10 trajectories)	–	~30 min
MCTS optimization	$M \times \mathcal{K} \times h$ expansions (4× parallel)	~25 min	~28 min
Total KB construction		~1h 25min	~1h 58min
Evaluation (369 episodes)	$369 \times \sim 2.5$ min/episode	~15h 20min	
KB as % of total		~1.3%	~1.8%

Table 7: End-to-end cost breakdown for OSWORLD. The hindsight relabeling phase adds ~30 minutes to KB construction, and the slightly larger concept set increases MCTS time marginally. Even with BREW-HR, KB construction accounts for less than 2% of the total compute budget.

The hindsight relabeling phase adds approximately 30 minutes: ~5 minutes for summarizing 10 failing trajectories and generating reframed tasks (LLM calls), and ~25 minutes for re-executing the agent on the 10 reframed tasks. The subsequent MCTS phase is only marginally more expensive for BREW-HR (~28 min vs. ~25 min) because the additional meta-concepts (27 vs. 23) represent a modest increase in parallel search workload. Overall, BREW-HR’s KB construction accounts for ~1.8% of the total compute budget, compared to ~1.3% for BREW; both negligible relative to the cumulative evaluation cost, while yielding a net reduction of ~15% in execution-time API calls through improved agent efficiency.

A.4.6 Scalability Considerations

The number of meta-concepts $|\mathcal{K}|$ is the primary factor governing MCTS cost. In all evaluated benchmarks, $|\mathcal{K}|$ remains below 30 due to effective semantic deduplication, even under BREW-HR, which enriches the insight pool. For domains requiring substantially more concepts, several strategies can maintain tractability:

- **Increased parallelism:** Since EG-MCTS optimizes each concept independently, scaling to $|\mathcal{K}| = 1,000$ requires proportionally more parallel processes but no algorithmic changes.
- **Concept prioritization:** Concepts can be ranked by retrieval frequency or reward variance, and MCTS budget can be allocated preferentially to high-impact concepts.
- **Hierarchical clustering:** For very large concept sets, a two-level hierarchy (concept groups $\rightarrow$ individual concepts) can reduce the effective search space.
- **Selective relabeling:** Under BREW-HR, the balanced sampling strategy (Algorithm 6) ensures that relabeling cost scales with $\frac{1}{2}|\mathcal{Q}_{\text{train}}|$ rather than with the full training set. For larger training sets, importance sampling can further focus relabeling on the most informative failures.

A.5 BREW Configurations

Base LLM Configuration For all BREW algorithm steps, we use the OpenAI GPT-4.1-2025-04-14 model as the underlying language model. To balance exploration and stability, we set the temperature to 0.7 for the IntegAgent component to encourage diversity in sampled completions, while all other calls use a temperature of 0.1 for deterministic behavior. The search process employs an expansion width of $e = 3$, a maximum search depth of $k = 3$, and a maximum of $n = 10$ iterations. Reward signals are weighted equally across correctness and retrieval relevance, with $\lambda_{\text{corr}} = \lambda_{\text{ret}} = 0.5$.

A.6 Experimental Setup

A.6.1 Baseline Methods

We compare BREW against two common reasoning baselines. In-Context Learning augments the input prompt with successful trajectories from related tasks, enabling the model to benefit from relevant prior examples without additional fine-tuning.

Benchmark	Baseline	Agent-Mem	Cognee	BREW	BREW-HR
OSWorld	50.00	76.00	81.50	78.00	84.00
τ^2 Bench	50.00	78.33	72.00	78.33	72.67
SpreadsheetBench	50.00	63.83	71.40	83.39	77.49

Table 8: Train-set accuracy across methods.

A.6.2 Benchmark Specifications

OSWORLD: Computer-Use Automation. *Dataset Overview.* OSWORLD (Xie et al., 2024) comprises 369 real-world computer-use tasks spanning 10 distinct applications. The benchmark is divided into train and test sets, with the distribution of tasks across domains shown in Table 9.

Agent Specifications. The UI-Tars-7B variant is a 7B-parameter multimodal transformer fine-tuned for graphical user interface understanding. It operates over an action space of PyAutoGUI commands (e.g., click, type, and key presses). The agent integrates a retrieval module that queries a task-relevant knowledge base using the user-provided description, with the top three retrieved items added to the system prompt. Inputs to the model consist of a screenshot of the active UI paired with the natural language task description.

The GTA1-7B configuration adopts a two-agent architecture, consisting of a planner and a grounding module. The planner (GTA-1-7B) generates the high-level action sequence, while the grounding module (OpenAI O3) verifies and refines each action before execution. Knowledge retrieval is incorporated differently for each component: the planner performs a single retrieval at the start of execution, which is persisted in its prompt, whereas the grounding module performs dynamic retrievals at each verification step.

Evaluation Protocol. Evaluation uses 134 task-specific scripts designed for automated verification. Success criteria include file state checks (e.g., validating .xlsx or .docx outputs), UI element validation to confirm correct interaction, and process completion checks to ensure that the intended automation sequence was executed successfully.

Domain	Test	Train
Calc	45	2
Chrome	44	2
Writer	21	2
Gimp	24	2
Impress	45	2
Os	22	2
Thunderbird	13	2
Multi-apps	99	2
VLC	15	2
VSCode	21	2
Total	349	20

Table 9: Test and Train samples across different domains in OSWORLD.

τ^2 -Bench: Interactive Tool Usage. *Dataset Overview.* τ^2 -Bench (Barres et al., 2025) extends τ -Bench by introducing bidirectional tool-calling capabilities. The dataset covers multiple service-oriented domains, with domain-level task distributions summarized in Table 11.

Domain Characteristics. The benchmark spans several domains with distinct task characteristics. The Telecom domain focuses on connectivity troubleshooting, plan modifications, and service activation workflows. The Retail domain includes order processing, return handling, and inventory queries. The Airline domain emphasizes booking modifications and policy-compliant rescheduling scenarios.

Interaction Settings. Two interaction modes are defined. In Easy mode, a human proxy (implemented via GPT-4.1) provides detailed guidance to the agent. The knowledge base is built exclusively from Easy mode trajectories, ensuring high-quality demonstrations for learning. In Hard mode, human intervention is minimized. The knowledge base combines both Easy and Hard trajectories, testing the agent’s robustness to underspecified or noisy instructions.

Evaluation Criteria. Task success is measured using domain-specific verification procedures. These include database state checks to validate final outcomes, status checks for confirming service or connection state, natural language verification to ensure correct confirmation statements appear in dialogue, and action matching to confirm that all required steps are completed. Each domain uses a tailored subset of these checks (e.g., Telecom relies primarily on status checks).

SPREADSHEETBENCH: Real-World Spreadsheet Manipulation. *Dataset Overview.* SPREADSHEETBENCH (Ma et al., 2024) consists of 912 instructions collected from four major Excel forums and blogs. Each instruction is paired with spreadsheets reflecting authentic, complex user scenarios, often containing multiple tables and non-standard relational structures. The dataset totals 2,729 test cases, averaging three per instruction. A breakdown of cell-level and sheet-level manipulations is shown in Table 12.

Task Settings. The benchmark defines two dimensions of evaluation:

Error Type	Baseline	Cognee	Agent-Mem	BREW
Wrong info	26	<u>28</u>	24	16
Wrong argument	22	<u>20</u>	<u>24</u>	14
Wrong decision	28	<u>30</u>	<u>26</u>	18
Partially resolve	30	<u>32</u>	28	24
Total	106	<u>110</u>	102	72

Table 10: Error distribution (%) on τ^2 -Bench Retail. $\downarrow$ is better. Best in **bold**, worst underlined.

Domain	Test	Train
Telecom	105	7
Retail	105	7
Airline	44	6
Total	254	20

Table 11: Task-wise breakdown for τ^2 -Bench with assumed 2-shot training samples per domain.

- **Granularity:** Instructions involve either *cell-level* manipulations (specific ranges such as D2:D6) or *sheet-level* manipulations (entire tables or multi-sheet updates).
- **Evaluation:** Performance is measured using an Online Judge (OJ)-style protocol. The *soft* setting (IOI-style) awards partial credit when only some test cases are solved, while the *hard* setting (ICPC-style) requires solutions to succeed on all test cases.

Agent Configuration. We evaluate o4-mini using a function-calling agent connected to a single Python execution tool. The agent translates natural language instructions into Python code for spreadsheet manipulation (e.g., modifying cells, applying formulas, restructuring tables). After each tool call, all formulas in the spreadsheet are recalculated to ensure consistency before proceeding to the next step. This setup provides a controlled environment to assess reasoning, code generation, and execution robustness across diverse spreadsheet tasks.

Granularity	Instructions	Test Cases
Cell-Level	329	986
Sheet-Level	583	1,743
Total	912	2,729

Table 12: Cell-level vs. sheet-level distribution in SPREADSHEETBENCH.

A.6.3 KB Construction and Retrieval Details

Training Data Collection

- **OSWORLD:** 20 successful trajectories (2 per application domain) and 10 for evals.
- **τ^2 -Bench:** 20 trajectories balanced across domains and difficulty settings and 10 for evals.
- **SPREADSHEETBENCH:** Uniformly sample 30 trajectories for training and 10 for evaluation.

All numbers are reported on the remaining train set.

Retrieval Strategy

- **Query Formation:** For each task we take in the seed Natural Language query as the retrieval query.
- **Retrieval Count:** We take top-3 documents for all the retrieval steps.
- **Integration Point:** For SPREADSHEETBENCH and OSWORLD we insert retrievals in the system prompt augmentation. For τ^2 -Bench we perform retrieval after each user interaction.

A.7 Analysis and Ablations

A.7.1 Exploration of MCTS Parameters

We evaluate OSWORLD on two different MCTS parameter settings.

- **Increased Depth:** We keep maximum width of the tree as 3 and depth as 10 with max number of iterations as 25. We observe that the knowledge base over-optimizes on the train set leading to a poorer performance on test set.
- **Increased Width:** We reverse the parameters where depth is 3 and maximum width is 10 with max iterations 25. We observe many different styles of KBs are generated storing very similar information; these different styles lead to varied performance on both eval and test set, notifying the importance of state search.

We report these numbers in Table 13.

	Baseline	max_width=3, max_depth=3	max_width=3, max_depth=10	max_width=10, max_depth=3
OSWORLD Verified	53.30	56.80	52.69	58.50
τ^2 -Bench	56.63	59.14	55.20	60.80
SPREADSHEETBENCH	44.30	46.80	43.20	47.90

Table 13: Ablation of MCTS tree parameters across benchmarks. Increasing width improves performance consistently, while deeper trees without sufficient breadth degrade results.

A.7.2 Generalizability of BREW

To assess the generalizability of BREW, we conduct experiments on the τ^2 -Bench dataset by varying the proportion of training data while keeping a fixed held-out evaluation set comprising 50% of the data. Specifically, we incrementally increase the amount of training data from 0% to 50% in steps of 10% and report performance on the held-out portion.

As shown in Figure 4, model performance improves steadily as more training data is provided, indicating that BREW effectively leverages additional supervision. Notably, the performance gain begins to saturate beyond 40% of the available training data, suggesting diminishing returns at higher data scales.

A.7.3 SPREADSHEETBENCH Qualitative Analysis

We identified 109 improvement cases where the BREW KB-enhanced system passed the first test case, the baseline system failed the first test case, and both systems attempted the same task with identical inputs except the KB. Some common failure patterns identified by BREW:

Header Detection and Data Structure Preservation. Spreadsheet data often lacks explicit headers or contains irregular header structures. Baseline trajectories frequently misinterpret the first data row as column names, corrupting the entire dataset structure.

Case Study – Task 280-17. **Task (Simplified):** Remove duplicate rows while keeping last occurrence of each value in column 1.

Baseline Approach:

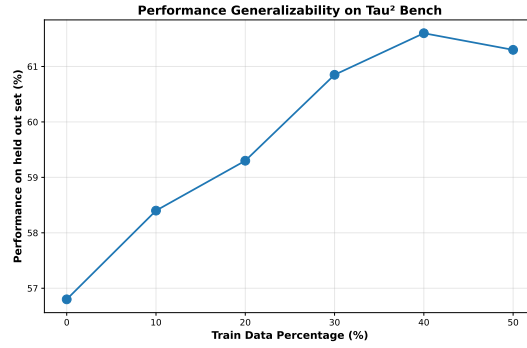


Figure 4: **Performance generalizability on τ^2 -Bench.** The plot shows the relationship between the proportion of training data and performance on the held-out set. BREW exhibits consistent gains with increased training data, demonstrating strong generalization behavior.

```
# Default pandas behavior
df = pd.read_excel(input_path, sheet_name='Sheet1')
last_indices = df.groupby(1, sort=False).apply(lambda grp: grp.index[-1])
filtered_df = df.loc[last_indices.values]
filtered_df.to_excel(output_path, index=False)
```

The baseline implementation used default pandas header detection, which promoted the first data row ("DATA 1", 1) to become column headers, fundamentally altering the dataset structure.

KB-Enhanced Approach:

```
# Explicit header handling guided by KB documentation
df = pd.read_excel(input_path, sheet_name='Sheet1', header=None)
target_col = df.columns[1]
df_filtered = df.drop_duplicates(subset=[target_col], keep='last')
df_filtered.to_excel(writer, sheet_name=name, header=False, index=False)
```

The KB system's "Header Extraction" documentation guided the conversation toward explicit header validation. The system probed the data structure using `df.columns.tolist()`, discovered the header issue, and applied appropriate parameters.

Impact: The difference is structural—the baseline output had incorrect data alignment due to header misinterpretation, while the KB output preserved the original data relationships.

Formula Quality and Error Resilience. Complex Excel formulas can require domain-specific knowledge about function behavior, particularly regarding blank cell handling and reference types.

Case Study – Task 55421. **Task (Simplified):** Create conditional logic for medical appointment scheduling based on multiple criteria including blank date fields.

Baseline Formula (Incorrect):

```
=IF(AND(D2="No Show",E2=""), "CALL PT", "")
```

BREW KB-enhanced Formula (Correct):

```
=IF(AND($D2="No Show", ISBLANK($E2)), "CALL PT", "")
```

The baseline approach uses string comparison (`E2=="`) which fails to properly detect blank cells in Excel. The KB system’s validation guidance led to using the `ISBLANK()` function and absolute references (`$D2`, `$E2`), resulting in robust formulas that handle edge cases correctly.

Impact: This pattern appeared across multiple formula-intensive tasks, with KB guidance consistently leading to more resilient Excel function usage.

Precise Range Targeting and Scope Compliance. Spreadsheet tasks often specify exact cell ranges for modifications. Exceeding these boundaries violates task constraints and can corrupt unrelated data.

Case Study – Task 12864. **Task (Simplified):** Use INDEX/MATCH formulas to populate Sheet2!B2:B12 with lookups from Sheet1.

Scope Violation:

The baseline system wrote to both columns B and C:

Baseline Implementation (Scope Violation)

```
# Writes to both B and C columns, violating answer_position
row[1].value = deal_lookup.get(existing_field, "") # Column B
row[2].value = type_lookup.get(existing_field, "") # Column C
```

The KB system strictly adhered to the specified range:

KB Implementation (Compliant)

```
# Writes formulas only to B2:B12 as specified
formula = f"=INDEX(Sheet1!$B:$B,_MATCH($A{row},_Sheet1!$D:$D,_0))"
ws2.cell(row=row, column=2).value = formula # Only column B
```

Additional Differences:

- **Implementation approach:** Baseline used Python dictionaries; KB used actual Excel INDEX/MATCH formulas as requested.
- **Range verification:** KB included diff checking to confirm only target cells changed.
- **Documentation reference:** KB conversation explicitly mentioned “Column Selection” guidance.

Systematic Validation and Verification. Spreadsheet manipulation errors often cascade, making post-execution validation critical for reliability. KB-enhanced conversations consistently included verification steps:

- Diff generation to confirm only target cells changed
- Output file reloading and inspection
- Formula validation through sample execution
- Explicit reference to “Difference in State” documentation

This validation mindset prevented many edge case failures that appeared in baseline implementations.

Discussion. Automated spreadsheet manipulation presents unique challenges due to the diversity of data structures, formatting conventions, and domain-specific requirements. While large language models demonstrate general competency in coding tasks, they often struggle with the nuanced requirements of spreadsheet operations such as header detection, precise range targeting, and formula construction. Our analysis demonstrates that knowledge base impact extends beyond general process improvement. The documented cases show specific technical guidance that directly prevents implementation errors. For

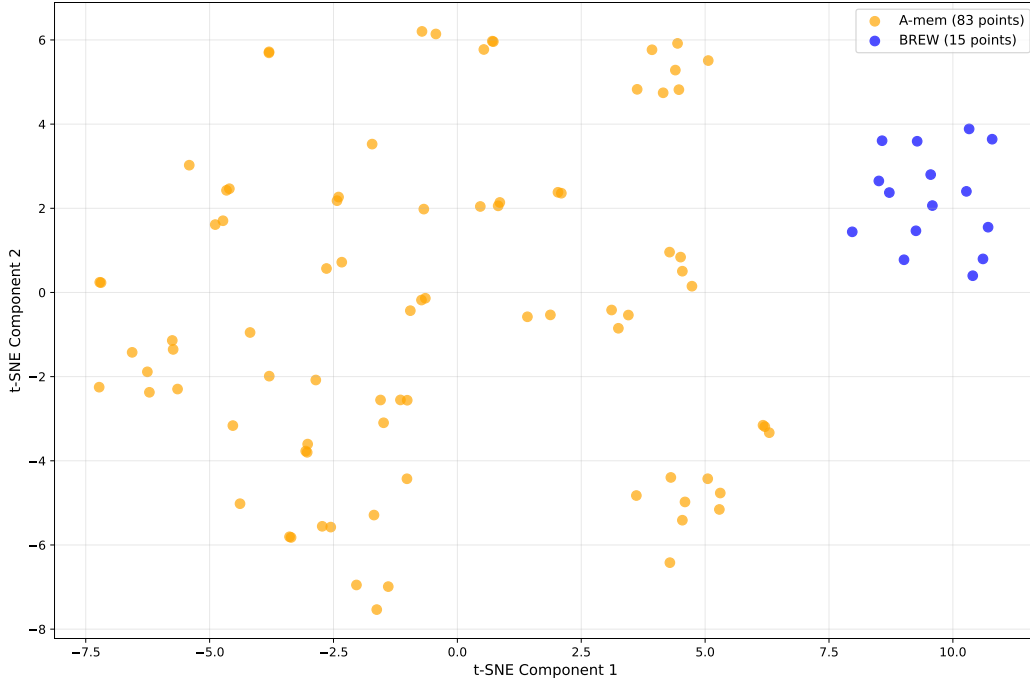


Figure 5: t-SNE plot of knowledge learned by BREW (blue), an experientially learning algorithm, and A-mem (yellow), an agentic tool-based memory storage that relies on the LLM to take memory save actions, on SPREADSHEETBENCH.

instance, the distinction between `E2=="` and `ISBLANK(E2)` represents domain-specific Excel knowledge that a general-purpose system might lack.

Figure 5 shows the t-SNE clustering of knowledge learned by A-mem and BREW. The BREW KB is highly structured and domain-adaptive (see Appendix A.8.3 for the learned KB). The embedding was done using text-embedding-3-large, where every A-mem entry and every keyword heading in BREW represents a point in the t-SNE embedding space.

The t-SNE visualization of the KB concept embeddings reveals that the knowledge learned by the BREW framework is semantically coherent, forming a dense cluster. This is in stark contrast to the sparse and diffuse memory representation of the A-mem baseline, suggesting that BREW’s distillation process produces a more structured and less noisy knowledge base. Analyzing the KB content further reveals that the structure of this knowledge is highly adaptive to the agent’s environment. For SPREADSHEETBENCH, the KB is dominated by abstract formula logic and declarative facts. For OSWORLD, it is primarily procedural, containing step-by-step instructions for GUI navigation. For τ^2 -Bench, the KB contains a blend of high-level strategy and procedural tool-use instructions, reflecting the dual demands of conversational reasoning and API execution. This demonstrates that the agent is not simply memorizing trajectories but is building a structured, internal model of its environment tailored to the specific challenges it faces.

A.7.4 Qualitative Analysis of BREW-Generated Knowledge Bases

This section presents a comprehensive qualitative analysis of knowledge bases generated through the BREW technique applied to two distinct agent training environments: OSWORLD and τ^2 -Bench. The analysis examines knowledge representation patterns, procedural sophistication, and domain-specific learning characteristics extracted from CUA agent behaviors, providing insights into the effectiveness and scope of knowledge distillation techniques across diverse task environments.

Base Structure and Organization. *Schema Consistency and Evolution.* Both knowledge bases demonstrate consistent structural schemas, though adapted to their respective domains. The OSWORLD KB employs a four-part schema (contextual triggers, procedural steps, extended capabilities, concrete instantiation), while the τ^2 -Bench KB extends this to a five-part structure, adding explicit purpose rationale (“Why to use it”). This evolution suggests that BREW adapts its extraction patterns to domain-specific requirements—conversational commerce demands explicit justification for actions due to customer interaction contexts.

Taxonomic Organization Principles. The OSWORLD KB reveals a **capability-based taxonomy** organized around computational tasks: file operations, document processing, inter-application workflows, and data visualization. Each category represents a distinct computational domain with specific tool requirements and interaction patterns. In contrast, the τ^2 -Bench KB employs a **lifecycle-based taxonomy** structured around transactional states: order creation, modification, fulfillment, and post-delivery operations. This organizational difference reflects fundamental domain characteristics—desktop automation focuses on tool orchestration, while conversational commerce centers on process management.

Hierarchical Task Decomposition. Both KBs demonstrate sophisticated hierarchical reasoning, but through different decomposition strategies. OSWORLD exhibits **technical decomposition**, breaking complex operations like “Create Charts from Data” into constituent technical steps (data selection, chart insertion, customization, formatting). τ^2 -Bench shows **process decomposition**, structuring operations like order modification into authentication, validation, confirmation, and execution phases. This suggests BREW successfully identifies domain-appropriate decomposition strategies rather than applying uniform patterns.

Knowledge Boundary Definition. Both KBs explicitly encode operational boundaries, but through contrasting mechanisms. OSWORLD boundaries are **capability-constrained**—determined by available applications and system resources. τ^2 -Bench boundaries are **policy-constrained**—explicitly defined through “Deny Unsupported Request” patterns and escalation protocols. This difference highlights how knowledge extraction adapts to domain-specific constraint types.

Procedural Knowledge Grounding. *Context-Dependent Action Selection.* Both domains demonstrate sophisticated context awareness, but grounded in different environmental factors. OSWORLD exhibits **application-context sensitivity**, where identical operations (e.g., image insertion) require different procedures across LibreOffice Writer, Impress, GIMP, and Thunderbird. The agent learned application-specific affordances and interaction patterns rather than generic command sequences. τ^2 -Bench demonstrates **state-context sensitivity**, where available actions depend on order status (pending vs. delivered), payment methods, and authentication levels. This reveals learned understanding of business process constraints and temporal operation windows.

Error Prevention and Validation Workflows. Both KBs incorporate sophisticated error prevention mechanisms, but grounded in domain-specific failure modes. OSWORLD emphasizes **technical validation**: file integrity checks (“confirm the exported file opens correctly”), application state verification, and multi-step confirmation for irreversible operations. τ^2 -Bench emphasizes **transactional validation**: authentication cascades, confirmation dialogues with standardized templates, and explicit user consent protocols. The emergence of defensive programming practices across both domains suggests these represent fundamental principles of reliable agent behavior.

State-Dependent Decision Logic. The procedural knowledge in both domains demonstrates sophisticated state machine reasoning. OSWORLD exhibits **application state awareness**—understanding when applications are ready for input, when files are loaded, and when operations can be safely executed. Window management and application switching reveal learned understanding of desktop metaphors and resource constraints. τ^2 -Bench demonstrates **business process state awareness**—finite state machine reasoning where order lifecycle states determine available operations. The agent learned that pending orders enable modification while delivered orders unlock return workflows, indicating internalized understanding of business logic constraints.

Security and Authentication Grounding. While OSWORLD operates in a trusted desktop environment with minimal explicit security concerns, τ^2 -Bench reveals pervasive **authentication-first paradigms**. Nearly every transactional operation begins with identity verification through email, name, and zip code combinations. The KB demonstrates **graduated security reasoning**: information retrieval requires basic authentication while financial transactions trigger rigorous verification protocols. This contrast highlights how procedural knowledge adapts to domain-specific security requirements.

Cross-Application vs. Cross-Process Orchestration. OSWORLD demonstrates **technical orchestration**—coordinating multiple applications (Chrome, LibreOffice suite, File Manager, GIMP) to accomplish complex workflows. The “Navigate Between Applications” section reveals learned behaviors for window management, application switching, and resource coordination. τ^2 -Bench exhibits **process orchestration**—coordinating authentication, validation, confirmation, and execution phases across different operational contexts. Both forms of orchestration require sophisticated temporal reasoning and constraint management, but applied to different environmental complexity types.

Failure Mode Internalization. Both KBs reveal learned understanding of domain-specific failure modes. OSWORLD incorporates file validation, application crash recovery suggestions, and verification steps for critical operations. τ^2 -Bench includes explicit escalation protocols (“Transfer to Human Agent”), policy compliance mechanisms, and irreversibility warnings for financial operations. The consistent emergence of failure-aware procedures suggests that agents successfully internalize risk assessment and mitigation strategies during training.

Domain-Specific Communication Patterns. The procedural knowledge reveals distinct communication paradigms appropriate to each domain. OSWORLD procedures are **task-oriented** with minimal user interaction—focusing on efficient command execution and verification. τ^2 -Bench procedures are **dialogue-oriented** with standardized customer interaction templates, confirmation protocols, and expectation management communications. This adaptation demonstrates that BREW extracts not just procedural logic but domain-appropriate interaction modalities.

The cross-domain analysis reveals that BREW successfully extracts procedural knowledge that is both **structurally consistent** (following learnable organizational patterns) and **contextually grounded** (adapted to domain-specific constraints, failure modes, and interaction requirements). This dual capability suggests significant potential for knowledge transfer across related domains while maintaining appropriate domain-specific adaptations.

A.8 Exemplar Knowledge Bases

A.8.1 Knowledge Base Learned for OSWORLD

We showcase a small part of the knowledge base learned through BREW. This demonstrates three major parts on which each document is aggregated. These parts discuss when to use a piece of information, why to use the information, and how to use the information/tool.

```
## Search and Open Files

**When to use**: Locating documents, spreadsheets, images, or downloads for editing,
conversion, or attachment.

### How to Perform
- Open **File Manager (Nautilus)** from launcher or system dock
- Press `Ctrl + F` or click the search icon
- Enter part of filename, full name, or wildcard (*.pdf, `report`)
- Use right-click -> **Open With** to choose the desired application
- Use the sidebar to navigate to **Downloads**, **Documents**, or custom folders

### Additional Actions
- Right-click -> **Properties** to check modification date or file type
- Sort results by Date, Type, or Name from the top-right dropdown
```

```
- Use `F2` to rename files inline

### Example
- Task: "Edit the file titled `sales_report_march.ods`"
  - Search for `sales` in File Manager
  - Confirm `.ods` type and open with LibreOffice Calc

...

## Insert Images

**When to use**: Adding visual elements to documents, presentations, emails, or
templates.

### How to Perform
- Navigate to **Insert -> Image -> From File** (in Writer, Impress, Thunderbird)
- Select an image file (`.png`, `.jpg`, `.svg`) from the file dialog
- Use drag handles to resize; right-click -> **Wrap** or **Alignment** for layout

### Additional Actions
- In GIMP: **File -> Open as Layers** to insert image as a new layer
- Use drag-and-drop from file manager into open document windows
- Use **Format -> Image** to apply borders, shadows, or color corrections (in
  Writer/Impress)

### Example
- Task: "Insert the logo.png image into the title slide"
  - Open `.odp` file in Impress -> Go to Slide 1 -> Insert -> Image -> Select `logo.png`

...

## Export as PDF

**When to use**: Required submission format

### How to Perform
- Go to **File -> Export As PDF**
- Choose output folder (usually **Documents** or **Downloads**)
- Click **Save**, then confirm the exported file opens correctly

### Additional Actions
- In GIMP or Impress: choose **File -> Export As**, then select `.pdf` from format list
- Use **Save As** to preserve both editable and exported versions separately

### Example
- Task: "Export the flyer.xcf as a PDF"
  - Open in GIMP -> File -> Export As -> Rename to `flyer.pdf` -> Click Export
```

A.8.2 BREW Knowledge Base for τ^2 -Bench

BREW enables us to learn relevant information for τ^2 -Bench across the domains in a single knowledge base. This knowledge base is helpful to use relevant actions from the action pool.

```
### Additional Actions

* Inform the user:
  - Refunds via gift card = immediate.
  - Refunds via other methods = 5--7 business days.

### Example
```

```
* Task: "Cancel a T-shirt order placed yesterday"
* Validate: Status is `pending`
* Reason: "no longer needed"
* Confirm
* Execute tool call

# Exchange Delivered Order

**When to use**:
User wants to swap delivered items for a different variant (e.g., size or color).

**Why to use it**:
To fix sizing or option errors without needing a new purchase.

### How to Perform
- Authenticate user
- Confirm order status is `delivered`
- Get full list of exchange items
> "Please ensure all items for exchange are listed. This step can't be repeated."
- Ask for refund/payment method
- Confirm:
  > "You're exchanging item X for same product, different option. Proceed?"
- On confirmation:
  ```python
 request_exchange(order_id="45678", item_exchanges=[...], payment_method="paypal")
  ```

### Additional Actions
* Mention: An email will be sent with return instructions
* Validate that the new variant is from the same product

### Example
* Task: "Exchange red shirt for blue in Order #45678"
* Confirm all exchange items
* Confirm payment method for difference
* Execute tool call

### Example
* Task: "Show me my last 2 orders"
* Authenticate
* Retrieve and present info

# Deny Unsupported Request

**When to use**:
User asks for an unsupported action (e.g., cancel processed order, exchange to
different product type, help another user).

**Why to use it**:
To stay compliant with platform policy.

### How to Perform
- Politely reject:
  > "I'm sorry, but I can't process that request. It's outside the allowed scope."

### Example
* Task: "Cancel a processed order"
* Respond with denial message
# Transfer to Human Agent
```

```

**When to use**:
User needs help outside the assistant's permitted capabilities.

**Why to use it**:
To ensure user gets the right help from trained staff.

### How to Perform
- Make tool call:
  ```python
 transfer_to_human_agents()
  ```
- Then inform user:
  > "YOU ARE BEING TRANSFERRED TO A HUMAN AGENT. PLEASE HOLD ON."

### Example
* Task: "Delete a task"
* Deny deletion
* Transfer to human

```

A.8.3 BREW Knowledge Base for SPREADSHEETBENCH

```

Header Extraction
1. Detecting Header Rows
Overview:
To accurately identify header rows, scan the initial region of your dataset. This
process is crucial for mapping column information for further processing.

Approaches:
- Heuristic Checks:
  - Look for rows where all cells are strings (e.g., "Name", "Date", "Region", "Amount").
  - Identify rows with distinctive formatting such as bold text or background color.
- Example:
  | Name | Date | Region | Amount | |-----|-----|-----|-----| | John |
    2024-01-01 | North | 100 |
- Pattern Recognition:
  - Use regex to match typical header patterns, such as column names starting with
    uppercase letters.
  - Score candidate rows based on the likelihood of being headers.
- Multi-Table Sheets:
  - Detect gaps, empty rows, or separators indicating a new table.
  - Assign a Table ID to each detected table for later reference.

Edge Cases:
- Merge multi-row headers (e.g., "Sales" over "2024", "2025" becomes "Sales 2024",
  "Sales 2025").
- Fill in missing headers by inferring from context.

2. Assigning and Validating Headers
Overview:
Once headers are detected, assign them programmatically and ensure they match expected
schema and data types.

Implementation:
- Column Naming:
  - Set names in code, e.g., df.columns = ["Name", "Date", "Region", "Amount"].
- Schema Mapping:
  - Map headers to a standardized schema, using external files or user prompts.
- Example:
  - Raw header: "Amt"; Mapped header: "Amount"
- Quality Checks:
  - Detect duplicate or empty headers ("Date", "Date" becomes "Date_1", "Date_2").
  - Validate each column's expected data type.

```

3. Automation and Usability Enhancements

Overview:

Enhance usability and automation to streamline header extraction and user interaction.

Features:

- Freeze Panes:
 - Automatically freeze header rows in Excel for easier navigation.
- Highlighting:
 - Use colored formatting to visually distinguish headers.
- Example:
 - Yellow fill for header row.
- Documentation:
 - Log extraction logic and confidence scores for each detected header.
- Integration:
 - Build header extraction into ETL pipelines and record process metadata.

Block Detection

1. Identifying Block Boundaries

Overview:

Block detection segments data into logical units or tables.

Methods:

- Boundary Detection:
 - Find empty rows, repeated labels, or formatting changes.
- Example:

| | | | | | | | | | | | | |
|-------|--------|--|--------|--|-------|-------|------|-----|--|--|--|-----------------------------|
| Name | Amount | | ----- | | ----- | | John | 100 | | | | <-- Empty row indicates new |
| block | Name | | Amount | | | Alice | 200 | | | | | |
- Machine Learning:
 - Train classifiers to detect block boundaries based on cell patterns.

Advanced:

- Detect nested blocks or hierarchies using indentation or merged cells.
- Identify summary blocks with keywords like "Total" or "Summary".

2. Processing and Tracking Blocks

Overview:

Once blocks are detected, assign IDs and enable block-level analysis.

Actions:

- Block ID:
 - Assign unique IDs (e.g., Block_001, Block_002).
- Analysis:
 - Perform group-by or aggregation within each block.
- Example:
 - Sum "Amount" for Block_001: $100 + 150 = 250$

3. Additional Block Actions

Overview:

Enable modular analysis and reporting at the block level.

Features:

- Summary Rows:
 - Add computed totals/averages for each block.
- Export/Save:
 - Save blocks as separate files or sheets.
- Example:
 - Export Block_001 to "block1.csv"

Search for Values or Patterns

1. Search Execution Methods

Overview:

Efficiently locate specific values or patterns in your data.

Techniques:

- Manual Tools:
- Use Ctrl + F in Excel for quick lookups.
- Programmatic Search:
- Scan all cells using loops or vectorized code.
- Example:
- Find all instances of "North" in the "Region" column.
- Pattern Matching:
- Support exact, wildcard (*Total*), and regex (\d{4}-\d{2}-\d{2} for dates).

2. Recording and Highlighting Results

Overview:

Log and visualize search matches for user review.

Actions:

- Logging:
- Record coordinates (e.g., Sheet1, Row 3, Col "Region").
- Highlighting:
- Apply conditional formatting to search hits.

3. Advanced Search Scenarios

Overview:

Handle complex or large-scale search requirements.

Scenarios:

- Merged Cells:
- Search within merged cells or across multiple sheets.
- Export:
- Export found results for further analysis.
- Example:
- Export all rows containing "John" to "john_results.csv"

Writeback Results

1. Output Placement

Overview:

Choose where and how to insert results.

Options:

- Target Columns:
- Select existing or blank columns for output.
- Appending:
- Add new columns for flags, counts, or statuses.
- Example:
- Add "Approved_Flag" column next to "Status".

2. Writing and Styling Results

Overview:

Automate and style the output for visibility.

Methods:

- Formulas/Code:
- Use code (e.g., `ws.cell(row, col).value = result`) to insert results.
- Styling:
- Bold, borders, or colors for output cells.
- Example:
- Green fill for "Success", red for "Error".

3. Audit and Protection

Overview:

Maintain the integrity and traceability of results.

Measures:

- Lock Columns:
- Prevent edits to output columns.
- Timestamps/User Info:
- Add audit trail for writebacks.

- Example:
- "2024-06-01, User: admin"

Difference in State

1. Sheet Comparison

Overview:

Identify changes between input and output sheets.

Process:

- Load Sheets:
- Read both sheets into memory.
- Compare Cells:
- Detect differences by position and value.

2. Recording and Reporting Differences

Overview:

Log and report all detected changes.

Actions:

- Log Mismatches:
- Record cell coordinates and values.
- Example:
- Cell B3: "North" -> "South"
- Export Diff Report:
- List all detected differences for review.

3. Visualization and Automation

Overview:

Make changes visible and automate validation.

Features:

- Highlight Changes:
- Color code changed cells.
- Automate Checks:
- Integrate diff comparisons into test scripts.

Column Selection

1. Selection Criteria

Overview:

Choose relevant columns for analysis.

Methods:

- Labels/Indices:
- Select by name or position.
- Dynamic Rules:
- E.g., all numeric columns.
- Assign Roles:
- Example: "ID", "Date", "Metric"

2. Preparation and Validation

Overview:

Prepare columns for consistent use.

Actions:

- Rename/Relabel:
- Standardize column names.
- Validate Types:
- Ensure columns are of expected type.
- Example:
- "Date" column as datetime.

3. Reusability

Overview:

Save and reuse column selections.

Features:

- Presets:
- Save selection profiles.
- Downstream Use:
- Use validated columns in subsequent processes.

Filter Rows

1. Filtering Methods

Overview:

Refine your dataset with filters.

Techniques:

- Spreadsheet Tools:
- Use built-in filters.
- Code Logic:
- Filter with code (e.g., `df[df['Status'] == 'Approved']`).
- Multiple Criteria:
- Combine conditions (AND/OR).
- Example:
- Status = "Approved" AND Amount > 100

2. Helper Columns and Complex Filters

Overview:

Simplify filtering using helper columns.

Actions:

- Helper Columns:
- Compute intermediate flags.
- Document Logic:
- Record filtering rules for audit.

3. Post-Filter Actions

Overview:

Visualize and export filtered data.

Features:

- Highlighting:
- Grey-out filtered-out rows.
- Export:
- Save the filtered dataset.

Merge Tables

1. Key-Based Merging

Overview:

Combine tables using shared keys.

Techniques:

- Join Operations:
- Use VLOOKUP, JOIN, or code merges.
- Example:
- Merge "Customer_ID" from two tables.
- Align Data:
- Match on columns like "ID", "Name".

2. Stack-Based Merging

Overview:

Append tables when keys aren't needed.

Methods:

- Vertical Append:
- Combine rows from similar tables.
- Deduplicate:
- Remove duplicate records.

3. Tracking and Audit

Overview:

Track source and unmatched records.

Actions:

- Source Column:
- Add "Source" to indicate origin.
- Highlight Unmatched:
- Mark or export mismatched rows.

Pivot or Unpivot

1. Pivoting Data

Overview:

Summarize data using pivots.

Methods:

- PivotTables:
- Group by row/column dimensions.
- Example:
- Sum "Amount" by "Region".
- Aggregation:
- Choose SUM, AVG, COUNT, etc.

2. Unpivoting (Melting) Data

Overview:

Reshape data from wide to long format.

Techniques:

- Melt Operations:
- Convert columns into rows.
- Example:

```
| Year | Sales_2019 | Sales_2020 | |-----|-----|-----|
->
```

```
| Year | Sales_Year | Value |
```

- Flexible Restructuring:
- Selectively unpivot non-ID columns.

3. Post-Pivot Actions

Overview:

Prepare pivoted data for export.

Features:

- Flatten Pivot Table:
- Convert back to flat for further analysis.
- Reorder/Rename:
- Clarify pivoted fields.

Map with Lookup Tables

1. Mapping Techniques

Overview:

Standardize data using lookups.

Methods:

- Functions:
- Use VLOOKUP, merge with dictionaries.
- Code-to-Label:
- Example:
- Code "N" -> Label "North"

2. Application and Fallbacks

Overview:

Apply lookups and handle missing values.

Actions:

- Apply Mappings:

- Across selected columns.
- Handle Missings:
- Use defaults for missing codes.

3. Audit and Display

Overview:

Ensure mapping transparency.

Features:

- Cache Mappings:
- Store for repeated use.
- Display Codes/Labels:
- Show both for clarity.

Fill Missing Data

1. Choosing Fill Methods

Overview:

Impute missing data appropriately.

Techniques:

- Forward/Backward Fill:
- Fill gaps with prior/next value.
- Default Values:
- Use fixed placeholder (e.g., 0, "Unknown").
- Contextual Example:
- Dates: Fill missing month with last known month.

2. Application and Auditing

Overview:

Apply fills and flag for review.

Actions:

- Targeted Filling:
- Apply to specific columns/rows.
- Flag Filled Cells:
- Highlight for later review.

3. Documentation

Overview:

Keep fill logic transparent.

Features:

- Record Logic:
- Document assumptions and methods.
- Audit Trail:
- Track all changes.

Flag Rows or Cells

1. Defining Flag Rules

Overview:

Establish criteria for flagging.

Examples:

- Simple Rule:
- Flag where Amount < 0
- Complex Rule:
- Flag where Status = "Pending" and Amount > 1000

2. Applying Flags

Overview:

Insert flags and summarize.

Actions:

- Flag Column:
- Add "Flag" column with "Yes"/"No".

- Export Flagged Rows:
- Save for further inspection.

3. Advanced Flagging

Overview:

Use multiple criteria and document.

Features:

- Multi-Criteria:
 - Combine several rules for granular checks.
- Notes:
 - Document flagging rationale.

Sort Data

1. Setting Sort Criteria

Overview:

Organize data for analysis.

Options:

- Sort Columns:
 - By value, ascending/descending.
- Multi-Level:
 - E.g., sort by "Region", then by "Amount".

2. Applying Sorts

Overview:

Implement sorting programmatically or manually.

Methods:

- Spreadsheet Tools:
 - Built-in sort features.
- Code:
 - E.g., `df.sort_values(['Region', 'Amount'])`

3. Post-Sort Actions

Overview:

Finalize sorted data.

Actions:

- Renumber Rows:
 - Update indices.
- Highlight Extremes:
 - Mark top/bottom values.

Validate Data

1. Validation Checks

Overview:

Ensure data meets required standards.

Checks:

- Type:
 - Ensure numeric columns contain numbers.
- Range:
 - E.g., `"Amount" > 0`.
- Pattern:
 - Date columns match `YYYY-MM-DD`.
- Business Rule Example:
 - `"Start Date" < "End Date"`

2. Marking and Reporting

Overview:

Visualize and report errors.

Actions:

- Highlight Invalids:

- Color-code errors.
- Export Summary:
- Table of error counts and locations.

3. Integration in Workflow

Overview:

Make validation a routine part of processing.

Features:

- Pre-Processing Step:
- Validate before analysis.
- Automation:
- Integrate into data pipelines.

Split Sheets or Data

1. Defining Split Rules

Overview:

Segment data for modular analysis.

Methods:

- By Category:
- E.g., split by "Region".
- By Date Range:
- E.g., split by year.

2. Exporting Segments

Overview:

Save segments for separate use.

Actions:

- Export Files:
- "North_Region.csv", "South_Region.csv"
- Consistent Formatting:
- Ensure identical columns and styling.

3. Automation and Documentation

Overview:

Automate splitting and track provenance.

Features:

- Automation:
- Use scripts/macros for repeated splits.
- Documentation:
- Record rules and export logs.