DAEDALUS: BOOTSTRAPPING AGENT MEMORY FROM SELF-GENERATED TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents often lack the operational knowledge to act reliably in new environments, as they must discover specific tool behaviors or environment conventions on their own. Without memory of past attempts, they repeat the same mistakes across tasks, leading to more task failures and longer trajectories. To address this, agentic systems typically rely on human-written guidelines or on procedural memory built from training tasks and an oracle verifier, both of which require prior knowledge of the environment. We present DAEDALUS, a method for bootstrapping reusable agent memory from self-generated practice without existing tasks or oracle verifiers. DAEDALUS pairs two agents: an explorer that interacts with the environment to generate challenging yet solvable tasks, and a solver that attempts them. A heuristic is derived from each solver failure and accepted only after the solver repeatedly succeeds with that heuristic in context. These outcomes also provide feedback for the explorer to refine the difficulty of future tasks. The accepted heuristics are then consolidated into a memory bank for test-time use. Across AppWorld, τ^2-bench, and AutomationBench, DAEDALUS improves mean success rates by up to 15.9 percentage points and pass^5 by up to 2.2$\times$ over a no-memory baseline, and is competitive with methods using training tasks, at a lower inference cost than most. We show that performance gains already emerge with a small exploration budget, and that its heuristics also benefit agents from other model families. Our ablations further reveal that solver traces provide the key information needed to derive effective heuristics, while factorizing early discoveries makes exploration more cost-efficient. Beyond memory construction, we find that the tasks generated by DAEDALUS can serve as a proxy for benchmark tasks when ranking models by performance.

1 INTRODUCTION

LLM agents are now deployed to perform real-world tasks in a wide range of occupations and industries (Sajadieh et al., 2026; Massenkoff et al., 2026; Johnston et al., 2026). As they gain autonomy (McCain et al., 2026), they operate in increasingly complex environments with limited human guidance. Acting reliably in such environments requires practical knowledge that carries over across tasks, a need reflected in the rapid adoption of agent skills (Zhang et al., 2025a). Without procedural memory to retain this knowledge, an agent rediscovers its environment from task to task, leading to avoidable errors and longer, costlier executions (Ouyang et al., 2026).

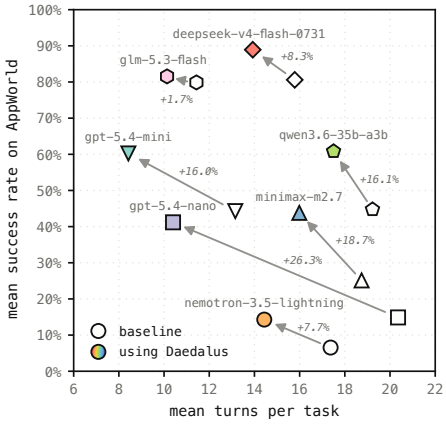


Figure 1: DAEDALUS memory from self-generated tasks improves agent success and efficiency across model families.

Procedural memory is typically built from past experience: an agent attempts tasks, and lessons drawn from its successes and failures are stored and injected in context at test time (Zhao et al., 2024; Allard et al., 2026; Zhang et al., 2026), which leaves model

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

weights untouched and applies to both open and closed models. This demands a curated set of training tasks drawn from the target task distribution, often paired with a verifier that scores each attempt. Building this supervision requires prior knowledge of the environment and substantial human effort, as the construction of agentic benchmarks illustrates (Trivedi et al., 2024; Barres et al., 2026). It is therefore unavailable for most environments, a limitation that is increasingly prevalent as more and more agentic environments are automatically generated without human supervision (Tu et al., 2026; Wang et al., 2026). Instead, experience can be gathered from task queries as they arrive at test time (Wang et al., 2025; Ouyang et al., 2026), but the agent then handles its first tasks without memory and improves only after failing on them. In both cases, memory is built only from exposure to the tasks it is meant to support.

In this work, we introduce DAEDALUS, a training-free method that builds environment-specific memory from self-generated tasks without curated tasks or an oracle verifier. DAEDALUS involves auxiliary agents that explore the environment to create solvable tasks and their success criteria, and a solver agent that attempts these tasks. The solver learns from its failures, as each one is turned into an in-context heuristic that guides its subsequent attempts. The relevance of a heuristic is validated only when it enables the solver to repeatedly succeed on a task it initially failed. Tasks that offer no such learning signal are refined to adjust their difficulty, and these refinements are condensed into task design guidelines for later sessions. Together, these components automate exploration and task creation in an agentic environment, calibrate task difficulty to the solver's capabilities, and produce memory items that are empirically tested.

On AppWorld (Trivedi et al., 2024), τ^2-bench (Barres et al., 2026), and AutomationBench (Shepard & Salimans, 2026), DAEDALUS improves success rates by up to 15.9 points over a no-memory baseline and raises pass^5 by 1.7 to 2.2×, at little to no extra inference cost. It outperforms PREPING (Choi et al., 2026), which also builds memory from self-generated tasks, and is competitive with existing methods that use the benchmark training tasks. These gains hold across model families, including when the memory is generated with one family and used by another, and can be achieved with only a small generation budget. Experiments show that validating lessons through the solver's attempts drives these results, whereas memory drawn from exploration alone hurts the agent. We also find that, at this scale, consolidating the bank and injecting it whole at the start of each task outperforms per-turn retrieval. Beyond memory, the generated tasks rank models consistently with the official benchmark test sets, which suggests they can serve as an evaluation proxy when no curated tasks exist. DAEDALUS thus bootstraps agent memory in new environments without human supervision, so that agents no longer discover them through trial and error at test time. Upon publication, we will release our code and experimental artifacts, including configurations, generated tasks, heuristic banks, and all agent trajectories, so that our experiments can be reproduced and verified.

2 Related Work

Memory for LLM agents. LLM-agent memory methods allow agents to accumulate reusable knowledge from past task attempts. Many store it as natural-language insights: ExpeL (Zhao et al., 2024) contrasts successful and failed trajectories, AutoGuide (Fu et al., 2024) makes such guidelines context-aware, and other systems generate reflections from failed attempts (Allard et al., 2026; Ouyang et al., 2026). These methods retain an insight based on the outcome of the trajectory it came from, without confirming that it helps the agent overcome the failure; DAEDALUS instead keeps an insight only once it has turned a failure into repeated success. Others build skill libraries, from Voyager's executable code skills (Wang et al., 2024) to natural-language skills refined through reward-based optimization (Mi et al., 2026; Yang et al., 2026; Huang et al., 2026), which requires oracle verifiers or weight updates. Memory can also be organized as reusable workflows (Wang et al., 2025), evolving playbooks (Zhang et al., 2026), or long-term memory stores (Fang et al., 2026; Chhikara et al., 2025). Most of these methods, however, collect experience from a predefined task set, which a new environment does not provide, whereas DAEDALUS generates its own.

Automated agentic task creation. Automatic exploration is widely used to synthesize agent training data in software, computer-use, and general-application environments (Shi et al., 2026; Ramrakhya et al., 2026; Dong et al., 2026), sometimes with curricula that adapt task difficulty to the agent's progress (Sun et al., 2026; Xue et al., 2026). Self-play extends this by training the model as both proposer and solver (Liu et al., 2026; 2025; Lu et al., 2026), rewarding tasks that are challeng-

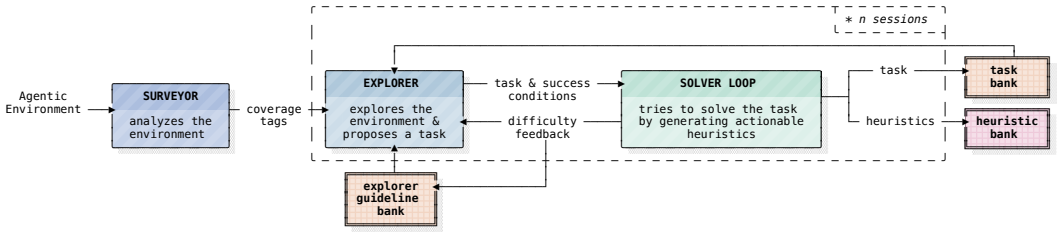


Figure 2: Overview of the DAEDALUS memory generation pipeline.

ing yet verifiable (Yue et al., 2026; Huang et al., 2026). In all these works, generated tasks serve to update model weights. Few methods generate tasks to build memory instead: Voyager (Wang et al., 2024) proposes tasks through an automatic curriculum, SkillWeaver (Zheng et al., 2025) practices self-proposed skills on websites, and PREPING (Choi et al., 2026), closest to our setting, builds a playbook from self-proposed tasks, updating it from each task's feasibility and completion scores. These methods mainly favor tasks that are feasible or broaden environment coverage, whereas DAEDALUS calibrates tasks to expose failures the agent can learn to overcome.

3 BUILDING MEMORY FROM SELF-GENERATED TASKS

Overview. We consider a base agent deployed in an agentic environment with no access to training tasks or an oracle verifier, and aim to improve its success rate without updating its weights. To this end, DAEDALUS spends a one-time budget of n practice sessions, bootstrapping a bank of textual heuristics that the base agent receives in context at test time (Figure 2). Each session runs two nested loops: in the outer loop, an Explorer agent proposes a task and adjusts its difficulty; in the inner *Solver loop* (Figure 3), a Solver agent attempts the task while an Extractor agent derives a heuristic from its failures. Besides the Solver, generation involves several other LLM agents, including the Explorer and the Extractor, which we call *auxiliary* agents. We describe the Solver loop first, then task calibration and consolidation. Appendix A formalizes the approach in pseudo-code.

Failure-to-success heuristic validation. Each task consists of an instruction and explicit success conditions. The Solver attempts the task from a freshly reset environment, and an LLM Judge checks the resulting trajectory against the success conditions; we measure the Judge's agreement with official benchmark verifiers in Section 5.1. When an attempt fails, the Extractor writes a heuristic from the instruction and the failed trajectory, or revises the heuristic written after an earlier failure, and the Solver retries with it in context (Shinn et al., 2023). The Extractor does not see which success conditions were missed, which discourages task-specific fixes in favor of generalizable guidance. The loop stops once the Solver succeeds N_s consecutive times or fails N_f times in total, with three possible outcomes: the heuristic is *accepted* if the Solver failed at least once before these N_s successes; the task is *too easy* if the Solver never failed; and it is *too hard* if the Solver reached N_f failures. Whereas prior methods validate memories by the outcome of the trajectory they were drawn from (Zhao et al., 2024; Fu et al., 2024; Allard et al., 2026; Ouyang et al., 2026) or by task feasibility (Choi et al., 2026), DAEDALUS thus tests the heuristic itself, and requiring N_s consecutive successes rather than a single one lowers the probability of accepting an ineffective heuristic after a success due to chance (Appendix C).

Calibrating tasks to the Solver. In each session, the Explorer interacts with the environment, proposes a realistic task, and completes it itself to establish feasibility (Shi et al., 2026; Ramrakhya et al., 2026). Because only accepted tasks yield heuristics, the Solver loop outcome doubles as a difficulty signal: when a task is too easy or too hard, the Explorer receives this outcome along with its previous attempt and revises the task to be harder or easier, up to N_r times. A session that exhausts its refinements contributes no heuristic. As in proposer–solver curricula (Yue et al., 2026), this feedback steers generation toward the edge of the Solver's capability, where failures are recoverable. In our main experiments, the Solver shares the base agent's model, and we show in Section 4 that the resulting bank also transfers to base agents from other model families. Figure 4 shows how an example session unfolds.

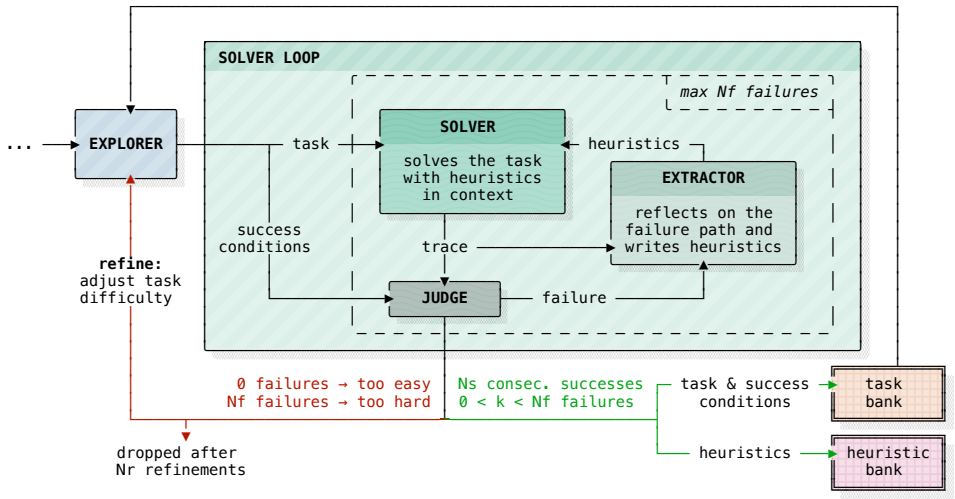


Figure 3: The Solver loop. After each failure, the Extractor writes or revises a heuristic and the Solver retries from a fresh environment state. The heuristic is accepted only after the *Solver* reaches the required streak of successes N_s.

Improving generation efficiency. Two components reduce generation cost (Section 5.1). Before the first session, a Surveyor explores the environment once and defines a target distribution of tasks across its areas, which the Explorer then follows to diversify the coverage of its tasks. In addition, after each session that required refinement, the refinement history is turned into *guidelines* that help the Explorer calibrate difficulty in later sessions.

Consolidation and deployment. After the last session, a Consolidator merges the accepted heuristics in a single LLM call, removing redundant or overlapping advice while preserving actionable guidance. The resulting bank is frozen and injected once into the base agent's context at the start of each test task; Section 5.2 compares this choice with per-turn retrieval.

4 Experiments

4.1 Evaluation Framework

Benchmarks. We run evaluations on three agentic benchmarks with different task types and environments: code-based app automation in AppWorld (Trivedi et al., 2024), conversational customer service in τ^2-bench (Barres et al., 2026), and business workflows with SaaS applications in AutomationBench (Shepard & Salimans, 2026). We evaluate agents on AppWorld's *normal* test split, whose tasks use the same apps as the training split. On τ^2-bench, we use only the *retail* domain, since *airline* contains ground-truth errors (Rabanser et al., 2026), and the dual-control setting of *telecom* would require simulating user-side tools during exploration. Due to compute costs, we restrict AutomationBench to its *Operations* split, whose higher public scores ensure a non-trivial baseline.

Baselines. We compare Daedalus against a no-memory baseline, which measures the capability of the base agent alone, and six memory methods. To our knowledge, PREPING (Choi et al., 2026) is the only existing method that also builds memory from self-generated tasks instead of training tasks. The other five build memory from the benchmark training tasks, and therefore have access to information that Daedalus does not. ExpeL (Zhao et al., 2024) and AutoGuide (Fu et al., 2024) contrast successful and failed retries of each training task to extract global insights and context-specific guidelines, respectively. ERL (Allard et al., 2026) and ReasoningBank (Ouyang et al., 2026) instead reflect on a single attempt per task, with the environment reward and an LLM judge as the respective feedback signals. ACE (Zhang et al., 2026) maintains an evolving playbook that it refines through incremental updates. Finally, to isolate the memory mechanism from task generation, we evaluate a Daedalus-curated variant that runs our Solver loop on the same training tasks.

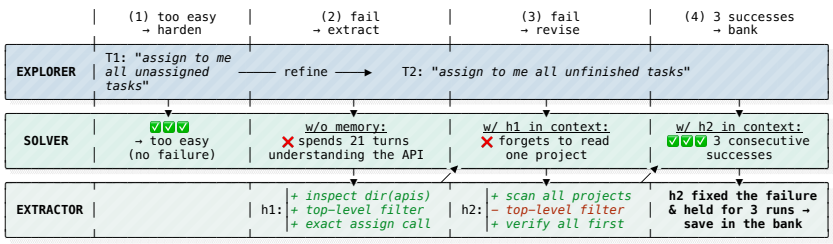


Figure 4: Timeline of AppWorld generation session 78. The Explorer first proposes a task that the Solver completes without failure, so the Explorer refines it to be harder. On the refined task, two failures yield successive heuristics h_1 and h_2. With h_2 in context, the Solver succeeds three consecutive times, validating its effectiveness; h_2 is therefore accepted into the memory bank. Appendix F.2 shows the session in greater detail.

Experimental setup. We split the public tasks of each benchmark into training tasks, from which we build reusable memory, and test tasks, on which we evaluate all methods. For AppWorld and τ^2-bench retail, we use the official splits (90/168 and 74/40 training/test tasks). For AutomationBench Operations, we split the 100 tasks 30/70, stratified by the number of services each task involves. Memory generation never accesses test instances and only uses environment states available to the baselines using training tasks (Appendix E).

To match the number of tasks available to the baselines, we run one DAEDALUS generation session per training task. The Explorer may refine each task up to $N_r = 5$ times, and the Solver loop stops after $N_f = 8$ failures. The loop accepts a heuristic after $N_s = 3$ consecutive successes. We analyze these limits and the effect of accepted heuristics on their own tasks in Appendix C. We set these values from a few preliminary generation sessions on AppWorld and reuse them on τ^2-bench and AutomationBench without tuning. The solver harness is a basic agent loop that executes Python code in AppWorld and uses standard function calls elsewhere, with no capabilities beyond what each environment natively provides.

All methods run a solver agent on tasks and delegate memory generation and retrieval to *auxiliary* agents, such as the Surveyor, Explorer, Judge, Extractor, and Consolidator in DAEDALUS. At test time, we evaluate a *base agent* that receives the resulting memory in context. We select two LLMs from the same family: the larger one for all auxiliary agents, and the smaller one for the solver and the base agent. By default, we use GPT-5.4 / GPT-5.4-mini on AppWorld and τ^2-bench, and GPT-5.6 Terra / GPT-5.6 Luna on AutomationBench, which is a more challenging benchmark. Appendix E lists the full configuration of each method.

For each method, we build one memory bank and average results over five independent inference runs, reporting the mean success rate (MSR) and pass^5 (Yao et al., 2025) with their standard errors. We also report the inference cost for one evaluation run at OpenAI standard-tier API prices, assuming perfect prompt caching.

4.2 MAIN RESULTS

DAEDALUS is the strongest method without training tasks. DAEDALUS raises the MSR of the base agent by 15.9, 10.0 and 4.3 points over the no-memory baseline on AppWorld, τ^2-bench and AutomationBench (Table 1). pass^5 increases by 1.7× to 2.2×, indicating that the method also improves consistency across runs. DAEDALUS also consistently outperforms PREPING on both metrics. On AppWorld and τ^2-bench, it does so at no extra inference cost over the no-memory baseline, mainly because the agent needs fewer turns (Figure 1).

DAEDALUS is competitive with methods that use curated tasks. Without any training task, DAEDALUS is within error margins of the best method using them in four of the six MSR and pass^5 columns, falling behind only ACE on AppWorld pass^5 and ExpeL on AutomationBench MSR. It is also robust across benchmarks as it never falls below the no-memory baseline, unlike ACE, ReasoningBank and AutoGuide. Given the same training tasks, DAEDALUS-curated ranks first within error margins on every benchmark and metric, at one of the lowest inference costs,

Table 1: Main results across five inference runs using a single memory bank per method and per benchmark. MSR denotes mean success rate, $\pm$ the standard error, and \$ the inference cost in USD per run, excluding memory generation. **Bold** marks the best result in each column, and shading the best result among methods without training tasks.

	AppWorld (normal)			τ^2-bench (retail)			AutomationBench (Operations)		
	MSR↑	pass^5↑	\$↓	MSR↑	pass^5↑	\$↓	MSR↑	pass^5↑	\$↓
No-memory baseline	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	3.2	$57.5_{\pm 1.6}$	$22.5_{\pm 6.6}$	**0.5**	$31.7_{\pm 3.2}$	$12.9_{\pm 4.0}$	1.0
Methods using training tasks									
AutoGuide	$44.0_{\pm 0.5}$	$17.9_{\pm 3.0}$	7.6	$61.5_{\pm 2.3}$	$27.5_{\pm 7.1}$	3.2	$35.4_{\pm 1.7}$	$18.6_{\pm 4.6}$	4.7
ReasoningBank	$48.0_{\pm 1.0}$	$19.6_{\pm 3.1}$	3.3	$64.0_{\pm 2.9}$	$32.5_{\pm 7.5}$	**0.5**	$19.1_{\pm 1.7}$	$2.9_{\pm 2.0}$	**0.7**
ERL	$58.6_{\pm 1.3}$	$29.2_{\pm 3.5}$	23.0	$64.0_{\pm 3.6}$	$30.0_{\pm 7.2}$	5.1	$33.4_{\pm 1.5}$	$15.7_{\pm 4.3}$	3.2
ExpeL	$59.0_{\pm 1.9}$	$33.3_{\pm 3.6}$	4.6	$67.0_{\pm 1.8}$	**37.5**$_{\pm 7.8}$	0.9	**42.6**$_{\pm 0.5}$	$21.4_{\pm 4.9}$	1.5
ACE	$60.5_{\pm 0.8}$	**41.1**$_{\pm 3.8}$	7.6	**70.5**$_{\pm 3.8}$	**37.5**$_{\pm 7.8}$	2.2	$26.3_{\pm 1.9}$	$10.0_{\pm 3.6}$	0.8
DAEDALUS-curated	**60.8**$_{\pm 0.9}$	$36.3_{\pm 3.7}$	**3.0**	$70.0_{\pm 3.6}$	$35.0_{\pm 7.5}$	0.6	$40.0_{\pm 2.3}$	**24.3**$_{\pm 5.1}$	1.0
Methods without training tasks									
PREPING	$56.0_{\pm 1.4}$	$25.6_{\pm 3.4}$	4.0	$60.0_{\pm 1.8}$	$25.0_{\pm 6.9}$	0.7	$34.9_{\pm 1.5}$	$15.7_{\pm 4.3}$	1.1
DAEDALUS	$60.2_{\pm 0.9}$	$32.1_{\pm 3.6}$	3.2	$67.5_{\pm 2.5}$	**37.5**$_{\pm 7.8}$	**0.5**	$36.0_{\pm 1.5}$	$21.4_{\pm 4.9}$	1.2

which shows that our memory mechanism is competitive independently of task generation. Self-generated tasks recover most of the benefit of curated ones, since DAEDALUS underperforms its curated variant by only 0.6, 2.5 and 4.0 MSR points, and the pass^5 gap is within error margins on all three.

Heuristics generated by one model family transfer to another. We investigate to what extent DAEDALUS generalizes to various model families, and whether a bank generated with one model family also helps base agents from another. We therefore run DAEDALUS on AppWorld with auxiliary / solver model pairs from the Qwen and DeepSeek model families, and evaluate each memory bank with each base agent. Table 2 shows that all nine gains are positive, indicating that a base agent does not need the bank of its own model family. For example, GPT-5.4-mini gains as much from the Qwen bank as from its own (+16.8 against +15.9). DeepSeek-V4-Flash gains less (3.0 to 8.3 points), as its no-memory baseline of 80.6% leaves less room for improvement. Figure 1 extends this result to more base agent LLMs. With DAEDALUS memory, they all reach a higher MSR in fewer turns, showing that our method benefits a wide range of models.

Table 2: Cross-family transfer of DAEDALUS heuristics on AppWorld. First row: MSR of each base agent (column) without memory. Others: MSR improvement when the base agent uses the DAEDALUS heuristic bank generated by an auxiliary / solver pair (row).

DAEDALUS *memory*		*Base agent*		
Auxiliary	Solver	GPT-5.4-mini	Qwen3.6-35B-A3B	DeepSeek-V4-Flash
No-memory baseline		$44.3_{\pm 1.1}$	$44.8_{\pm 1.3}$	$80.6_{\pm 1.6}$
GPT-5.4	GPT-5.4-mini	$+15.9_{\pm 1.4}$	$+16.1_{\pm 1.6}$	$+8.3_{\pm 1.8}$
Qwen3.8-Flash	Qwen3.6-35B-A3B	$+16.8_{\pm 2.4}$	$+29.3_{\pm 1.7}$	$+4.9_{\pm 2.0}$
DeepSeek-V4-Pro	DeepSeek-V4-Flash	$+11.4_{\pm 3.6}$	$+14.5_{\pm 1.3}$	$+3.0_{\pm 3.0}$

5 ANALYSIS AND ABLATIONS

This section examines each stage of DAEDALUS. For memory generation (Section 5.1), we ablate the pipeline components, validate the LLM judge against official benchmark verifiers, and vary the exploration budget and the auxiliary models. For inference (Section 5.2), we compare consolidating the bank and injecting it whole with retrieving heuristics at each turn. Finally, we test whether the generated tasks can serve as a proxy test set for ranking models (Section 5.3). Unless noted

Table 3: Cumulative ablation of the DAEDALUS generation pipeline on AppWorld. Each row adds its component to the preceding configuration; Accept. % is the percentage of 90 sessions ending with an accepted heuristic; # Refin. counts task revisions across sessions; Inf. $ and Gen. $ denote the inference and one-time memory generation costs, respectively.

	MSR↑	pass^5↑	Inf. $↓	Accept. %↑	# Refin.↓	Gen. $↓
No-memory baseline	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	3.2	–	–	–
Single Explorer (A)	$35.7_{\pm 0.8}$	$10.1_{\pm 2.3}$	3.5	–	–	5.2
+ Multi-session Explorer (B)	$38.7_{\pm 3.6}$	$11.3_{\pm 2.4}$	2.9	–	–	53.6
+ Explorer tasks and Solver traces (C)	$54.5_{\pm 2.8}$	$25.0_{\pm 3.3}$	**1.9**	–	–	39.9
+ Solver loop (D)	$59.0_{\pm 1.4}$	$30.4_{\pm 3.6}$	3.3	86.7	183	210.4
+ Explorer guideline bank (E)	$57.9_{\pm 2.1}$	$27.1_{\pm 3.5}$	3.2	91.1	161	185.3
+ Environment survey (DAEDALUS) (F)	$\mathbf{60.2}_{\pm 0.9}$	$\mathbf{32.1}_{\pm 3.6}$	3.2	**97.8**	**118**	109.7

otherwise, all analyses use AppWorld, with GPT-5.4 for the auxiliary agents and GPT-5.4-mini for the Solver and base agent.

5.1 MEMORY GENERATION

Pipeline ablation. We perform a cumulative ablation on AppWorld to assess how each component of DAEDALUS affects downstream performance and generation efficiency (Table 3). Starting with heuristics extracted from a single 100-turn Explorer trajectory (A), we switch to multiple 40-turn sessions with access to earlier heuristics (B). In (C), the Explorer proposes tasks for a Solver to attempt, and heuristics are extracted directly from all Solver trajectories, without retry-based validation. We add failure-triggered revisions and repeated-success validation in (D), guidelines derived from task refinements in (E), and a preliminary environment survey in (F), yielding the complete DAEDALUS pipeline. All multi-session variants use 90 sessions. Two findings emerge:

(I) Solver-grounded heuristics drive downstream performance. Exploration alone is insufficient: variant (A) underperforms the no-memory baseline, and the additional cost of multiple sessions does not allow (B) to overcome that loss. Two substantial jumps appear along the success axis. The first one (+15.8 MSR) comes from including Solver traces (C), showing that the qualitative content to derive heuristics lies in attempt trajectories. The second jump (+4.5 MSR) lands when validating heuristics with our Solver loop, which empirically removes noisy lessons from the bank, a result that is consistent with the scores of DAEDALUS-curated in Table 1.

(II) Task design guidelines and the environment survey improve generation efficiency. Guidelines in (E) carry refinement lessons into later sessions, helping the Explorer reach the target difficulty. They improve cost-efficiency rather than memory quality, reducing refinements from 183 to 161 and generation cost by 12%, while MSR changes by -1.1 points, within standard error (Table 3). The survey in (F) gives the Explorer a target task distribution over environment areas, reducing the exploration needed to select each task. Together, these components increase the share of sessions yielding an accepted heuristic and cut generation cost by 48% relative to (D). Figure 5 shows that Solver and Judge costs fall by 60% due to fewer refinements.

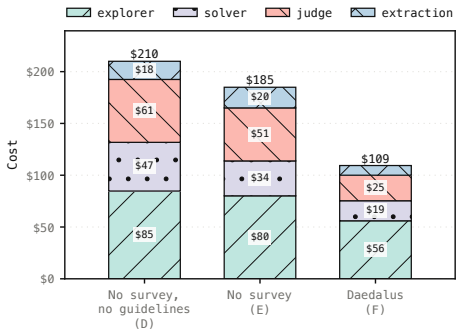


Figure 5: Generation cost by agent role for (D), (E), and the complete pipeline.

LLM judge validation. A mistaken judgment can misdirect task refinement or validate an unreliable heuristic. To assess our LLM judge, we compare it against benchmark-provided verifiers on official test sets. For each task, we let our Explorer inspect the environment and restate the benchmark's oracle requirements as natural-language success conditions, as it would for a task it generated. The judge then receives the task instruction, these conditions, and the Solver trajectory, but not the official verifier or its verdict. For each benchmark with N tasks, we judge one Solver trajectory

Table 4: Agreement between the LLM judge and official verifiers, and across five judge calls per trajectory. Precision and recall treat success as positive.

Benchmark	N	Precision	Recall	κ_{bench}	κ_{inter}
AppWorld	168	.943	.846	.807	.848
τ^2-bench	40	.889	.842	.749	1.000
AutomationBench	70	.863	.807	.729	.902

per task five times and measure agreement with the official verifier (κ_{bench}) and between repeated judgments (κ_{inter}). Table 4 shows substantial agreement with the official verifiers ($\kappa_{\text{bench}} > 0.7$) and consistent verdicts across calls ($\kappa_{\text{inter}} > 0.8$). Precision exceeds recall on every benchmark, so the judge leans toward rejecting successes rather than accepting failures, which is the safer direction for accepting heuristics.

Scaling with the exploration budget. We study how performance depends on the exploration budget by continuing a generation run up to 140 sessions and evaluating the memory at successive checkpoints (Figure 6). Five sessions already deliver more than half of the improvement reached at the 90-session peak. Beyond 90 sessions, MSR decreases even as the bank grows. A hierarchical consolidation, designed to scale to large banks, does not recover the loss (54.6 vs. 53.7 MSR): with a larger bank, consolidation misstates the scope of some heuristics, and such minor changes can have a large impact on specific parts of the test distribution (Appendix B). The early gains matter more in practice, since they make a useful memory affordable after only a few sessions.

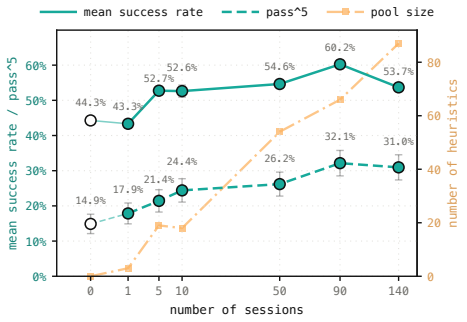


Figure 6: Scaling with the exploration budget on AppWorld, over a single generation run. Error bars are standard errors over five inference runs.

Effect of model selection. In our default setup, auxiliary agents run on a larger model than the Solver and base agent. To test whether DAEDALUS requires this asymmetry, we run every auxiliary agent with GPT-5.4-mini (Table 5). This setup still outperforms the no-memory baseline by 6.9 MSR and 5.3 pass^5 points, about half of the gain obtained with GPT-5.4. A stronger generation pipeline thus yields better memory, but DAEDALUS remains useful when a larger model is not available.

Table 5: Effect of auxiliary model choice.

Aux. model	MSR↑	pass^5↑	Gen. $↓
no memory	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	0
GPT-5.4-mini	$51.2_{\pm 1.6}$	$20.2_{\pm 3.1}$	67.2
GPT-5.4	$\mathbf{60.2}_{\pm 0.9}$	$\mathbf{32.1}_{\pm 3.6}$	109.7

5.2 MEMORY USE AT INFERENCE

Heuristics accepted in separate sessions often overlap, which raises two design questions at inference: whether to consolidate them into a compact bank, and whether to inject that whole bank or retrieve some heuristics at each turn. Injected at task start, the consolidated bank is both the best-performing and the cheapest option, improving MSR by 10.7 points over the unconsolidated heuristics while considerably shortening the context; deduplication alone yields no significant gain (Table 6). We then test whether retrieval can select more relevant heuristics per turn, comparing BM25, Qwen3-Embedding-4B (Zhang et al., 2025b), and a random control, each retrieving five heuristics per turn (Appendix D). All three improve over the no-memory baseline by up to 10 MSR points but do not differ significantly. A plausible reason is that useful heuristics are not necessarily semantically similar to the current step: retrieving them is an oblique retrieval problem, in which relevance is latent rather than expressed in the text (Tchuindjo et al., 2026). In line with this, retrieving more heuristics with BM25 improves success, but stays below whole-bank injection (58.2% at k=30; Figure 7). At this scale, injecting the whole bank once at task start is thus the most effective option we tested, while larger banks would call for retrievers suited to oblique queries.

Table 6: Consolidation and injection of the heuristic bank on AppWorld. **Bold**: best; underline: second best.

Policy	Dedup.	Consol.	MSR↑	pass^5↑	$↓
No memory	–	–	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	3.2
Top-5 per turn					
Random	✓	✓	$51.7_{\pm 1.5}$	$19.0_{\pm 3.0}$	4.1
BM25	✓	✓	$52.0_{\pm 1.3}$	$22.0_{\pm 3.2}$	3.8
Qwen3-Emb.	✓	✓	$\underline{54.3}_{\pm 1.6}$	$24.4_{\pm 3.3}$	3.7
Whole bank per turn					
All	✓	✓	$51.5_{\pm 1.2}$	$\underline{28.0}_{\pm 3.5}$	3.5
Whole bank at start					
All	✗	✗	$49.5_{\pm 1.0}$	$26.2_{\pm 3.4}$	4.9
All	✓	✗	$49.8_{\pm 2.5}$	$27.4_{\pm 3.5}$	4.8
DAEDALUS	✓	✓	$\mathbf{60.2}_{\pm 0.9}$	$\mathbf{32.1}_{\pm 3.6}$	**3.2**

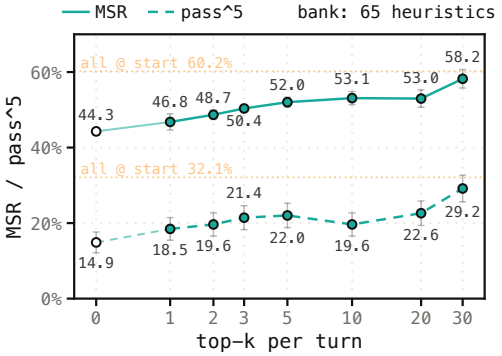


Figure 7: BM25 retrieval per turn with increasing k. Dotted lines: whole bank at start; k=0: no memory.

5.3 SELF-GENERATED TASKS AS EVALUATION PROXIES

Environments without training tasks usually lack a test set as well, which makes it hard to compare agents on them. We test whether the tasks generated by DAEDALUS can fill this role by evaluating nine models without memory on the 81 accepted tasks of the 90-session AppWorld run and on the AppWorld *normal* test split, with three runs per task set. Rankings are strongly aligned for both MSR (Kendall's $\tau = 0.83$, $p < .001$) and pass^3 ($\tau = 0.89$, $p < .001$), with 33 and 34 of 36 model pairs ordered consistently (Figure 8). Absolute success rates differ, as the generated tasks were calibrated to GPT-5.4-mini, but the relative order of models is preserved. These results suggest that DAEDALUS-generated tasks can serve as a proxy for ranking models when no curated test set is available.

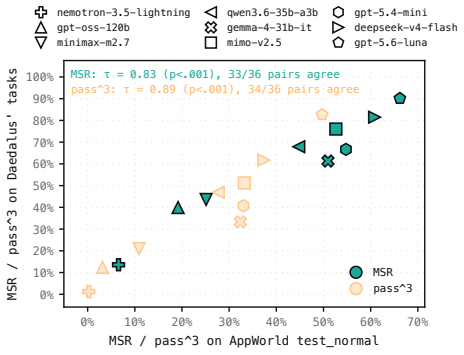


Figure 8: Performance on DAEDALUS-generated tasks versus the AppWorld test set, for nine models evaluated without memory.

6 CONCLUSION

Memory-based agents usually learn from curated tasks and an existing verifier, which a new environment does not provide. DAEDALUS removes the need for both: the agent builds its memory by practicing on calibrated tasks it generates itself, and keeps a heuristic only once it has turned one of the solver's failures into repeated success. Across three benchmarks, the resulting memory is the strongest among methods without training tasks and competitive with methods that require them, at marginal or no additional inference cost. The main reason behind this improvement is validation: heuristics drawn from exploration alone hurt the agent, whereas those grounded in its failures and confirmed empirically improve it. Additionally, DAEDALUS generates memory that transfers across model families, and tasks that rank models consistently with curated benchmarks, suggesting that it can also supply evaluation data where none exists.

On AppWorld, performance declined beyond 90 sessions as the memory grew, which raises open questions for scaling: how the composition and coverage of the bank vary across environments, how heuristics should be consolidated as the bank grows, and which retrieval mechanisms can best exploit large memories at inference time. DAEDALUS also builds its bank entirely before deployment and relies on an imperfect LLM judge; updating memory at test time and developing more reliable multi-judge validation schemes are natural next steps. We will release our code and all generated artifacts upon publication to encourage further work along these directions.

AI USE STATEMENT

We used generative AI tools to aid and polish the writing, draft sections of the paper, retrieve and discover related work, and support research ideation and execution, including code generation. We also used LLM agents to generate synthetic datasets: the tasks produced by DAEDALUS are the output of the method under study, serve as the source of its memory, and are used as an evaluation set in Section 5. Their generation is described in Section 3. We did not use generative AI tools to prove mathematical claims, as this work contains none. The authors carefully reviewed all AI-assisted work. Generated code was reviewed by the authors, including benchmark and baseline reimplementations, and each cited work was manually checked to confirm its existence. We take full responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This research raises no concerns under the ICLR Code of Ethics regarding human participants, personal data, or privacy. All experiments use publicly available benchmarks (AppWorld, τ^2-bench, and AutomationBench) whose environments and users are simulated. In our experiments, our agents act only inside sandboxed environments with no ability to interact with the outside world. The memory that DAEDALUS produces is plain text, which humans can inspect and edit before deployment.

REPRODUCIBILITY STATEMENT

The generation pipeline is described in Section 3 and formalized in Appendix A, and Appendix E gives the configuration of every method, including the models, splits and hyperparameters. Upon publication, we will release our code, all prompts, and the generated tasks, heuristic banks and agent trajectories, so that every experiment can be rerun or inspected.

REFERENCES

Marc-Antoine Allard, Arnaud Teinturier, Victor Xing, and Gautier Viaud. Experiential reflective learning for self-improving LLM agents. In *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, 2026.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik R Narasimhan. τ^2-Bench: Evaluating conversational agents in a dual-control environment. In *International Conference on Machine Learning*, 2026.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready AI agents with scalable long-term memory. In *European Conference on Artificial Intelligence*, 2025.

Yumin Choi, Sangwoo Park, Minki Kang, Jinheon Baek, and Sung Ju Hwang. PREPING: Building agent memory without tasks. *arXiv preprint arXiv:2605.13880*, 2026.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

Gaodan Fang, Vatche Isahagian, K. R. Jayaram, Ritesh Kumar, Vinod Muthusamy, Punleuk Oum, and Gegi Thomas. Trajectory-informed memory generation for self-improving agent systems. *arXiv preprint arXiv:2603.10600*, 2026.

Yao Fu, Dong-Ki Kim, Jaekyeom Kim, Sungryull Sohn, Lajanugen Logeswaran, Kyunghoon Bae, and Honglak Lee. AutoGuide: Automated generation and selection of context-aware guidelines for large language model agents. In *Advances in Neural Information Processing Systems*, 2024.

Siyuan Huang, Pengyu Cheng, Haotian Liu, Tao Chen, Yihao Liu, Jingwei Ni, Shijie Zhou, Ziyi Yang, Gangwei Jiang, Mengyu Zhou, et al. Skill self-play: Pushing the frontier of LLM capability with co-evolving skills. *arXiv preprint arXiv:2607.22529*, 2026.

Drew Johnston, David Holtz, Alex Martin Richmond, Christopher Ong, Prasanna Tambe, and Aaron Chatterji. The shift to agentic AI: Evidence from Codex. *arXiv preprint arXiv:2606.26959*, 2026.

Bo Liu, Chuanyang Jin, Seungone Kim, Weizhe Yuan, Wenting Zhao, Ilia Kulikov, Xian Li, Sainbayar Sukhbaatar, Jack Lanchantin, and Jason Weston. SPICE: Self-play in corpus environments improves reasoning. *arXiv preprint arXiv:2510.24684*, 2025.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, et al. SPADE: Self-play in adaptive synthetic executable environments. *arXiv preprint arXiv:2608.19197*, 2026.

Hongliang Lu, Yuhang Wen, Pengyu Cheng, Ruijin Ding, Jiaqi Guo, Haotian Xu, Chutian Wang, Haonan Chen, et al. Search self-play: Pushing the frontier of agent capability without supervision. In *International Conference on Learning Representations*, 2026.

Maxim Massenkoff, Eva Lyubich, Peter McCrory, Ruth Appel, and Ryan Heller. Anthropic economic index report: Learning curves. Anthropic Research, March 2026. URL `https://www.anthropic.com/research/economic-index-march-2026-report`.

Miles McCain, Thomas Millar, Saffron Huang, Jake Eaton, Kunal Handa, Michael Stern, Alex Tamkin, Matt Kearney, Esin Durmus, Judy Shen, Jerry Hong, Brian Calvert, Jun Shern Chan, Francesco Mosconi, David Saunders, Tyler Neylon, Gabriel Nicholas, Sarah Pollack, Jack Clark, and Deep Ganguli. Measuring AI agent autonomy in practice, 2026. URL `https://anthropic.com/research/measuring-agent-autonomy`.

Qirui Mi, Zhijian Ma, Mengyue Yang, Haoxuan Li, Yisen Wang, Haifeng Zhang, and Jun Wang. Skill-Pro: Learning reusable skills from experience via non-parametric PPO for LLM agents. In *International Conference on Machine Learning*, 2026.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026.

Stephan Rabanser, Sayash Kapoor, Peter Kirgis, Kangheng Liu, Saiteja Utpala, and Arvind Narayanan. Towards a science of AI agent reliability. In *International Conference on Machine Learning*, 2026.

Ram Ramrakhya, Andrew Szot, Omar Attia, Bogdan Mazoure, Anh Nguyen, Yuhao Yang, Zhe Gan, Harsh Agrawal, and Alexander Toshev. Scaling synthetic task generation for agents via exploration. In *International Conference on Learning Representations*, 2026.

Sha Sajadieh, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, Lapo Santarlasci, Juan Pava, Nestor Maslej, Russ Altman, Erik Brynjolfsson, et al. Artificial intelligence index report 2026. *arXiv preprint arXiv:2606.15708*, 2026.

Daniel Shepard and Robin Salimans. AutomationBench. *arXiv preprint arXiv:2604.18934*, 2026.

Dingfeng Shi, Jingyi Cao, Qianben Chen, Weichen Sun, Weizhen Li, Hongxuan Lu, Fangchen Dong, Tianrui Qin, King Zhu, Minghao Liu, Yuchen Jiang, et al. TaskCraft: Automated generation of agentic tasks. In *International Conference on Learning Representations*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Zeyi Sun, Ziyu Liu, Yuhang Zang, Yuhang Cao, Xiaoyi Dong, Tong Wu, Dahua Lin, and Jiaqi Wang. SEAgent: Self-evolving computer use agent with autonomous learning from experience. In *International Conference on Machine Learning*, 2026.

Diane Tchuindjo, Devavrat Shah, and Omar Khattab. OBLIQ-Bench: Exposing overlooked bottlenecks in modern retrievers with latent and implicit queries. *arXiv preprint arXiv:2605.06235*, 2026.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. ScaleEnv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. In *International Conference on Machine Learning*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. In *International Conference on Machine Learning*, 2026.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *International Conference on Machine Learning*, 2025.

Tianci Xue, Zeyi Liao, Tianneng Shi, Zilu Wang, Kai Zhang, Dawn Song, Yu Su, and Huan Sun. Autonomous continual learning for environment adaptation of computer-use agents. *arXiv preprint arXiv:2602.10356*, 2026.

Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. SkillOpt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*, 2025.

Zhenrui Yue, Kartikeya Upasani, Xianjun Yang, Suyu Ge, Shaoliang Nie, Yuning Mao, Zhe Liu, and Dong Wang. Dr. Zero: Self-evolving search agents without training data. In *Conference on Language Modeling*, 2026.

Barry Zhang, Keith Lazuka, and Mahesh Murag. Equipping agents for the real world with agent skills. Anthropic Engineering Blog, October 2025a. URL `https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills`.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI Conference on Artificial Intelligence*, 2024.

Boyuan Zheng, Michael Y. Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. SkillWeaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

A PSEUDO-CODE ALGORITHMS

We formalize the DAEDALUS memory generation phase (Algorithms 1–3), its supervised counterpart DAEDALUS-curated (Algorithm 4), and memory-augmented inference (Algorithm 5).

$\mathcal{E}$ is a collection of resettable environments, and $e \leftarrow \texttt{Reset}(\mathcal{E})$ selects an environment and restores its initial state. A task is $\tau = (u, \sigma)$, where u is its instruction and σ its success conditions. The agents are Surveyor, Explorer, Solver, Judge, Extractor, and Consolidator. The Surveyor returns a survey report $\mathcal{S}$ and coverage state $\mathcal{C}$. $\mathcal{T}$ contains accepted tasks, $\mathcal{H}$ accepted heuristics, and $\mathcal{G}$ task design guidelines. $\varnothing$ denotes absence and $[\,]$ an empty ordered history. The generation hyperparameters are n sessions, N_s consecutive successes, N_f cumulative failures, and N_r task refinements.

Sans-serif names denote prompted LLM calls (possibly multi-turn interactions with an environment), small-cap names algorithms defined below, and monospace names environment operations.

A.1 DAEDALUS GENERATION

Algorithm 1 DAEDALUS memory generation

Input: environments $\mathcal{E}$; hyperparameters n, N_s, N_f, N_r
Output: frozen heuristic bank $\mathcal{H}^\star$

$(\mathcal{S}, \mathcal{C}) \leftarrow \mathsf{Surveyor}(\mathcal{E})$
$\mathcal{T} \leftarrow [\,]$; $\mathcal{H} \leftarrow [\,]$; $\mathcal{G} \leftarrow [\,]$
for $i = 1, \ldots, n$ **do**
  $(\tau, h, o, \mathcal{L}) \leftarrow$ GENERATEHEURISTIC$(\mathcal{E}, \mathcal{S}, \mathcal{T}, \mathcal{G}, \mathcal{C}, N_s, N_f, N_r)$ ▷ Algorithm 2
  if $o = \texttt{accepted}$ **then**
    append τ to $\mathcal{T}$; append h to $\mathcal{H}$; update $\mathcal{C}$ with the tags of τ
  end if
  if $|\mathcal{L}| > 1$ **then**
    $\mathcal{G} \leftarrow \mathsf{CreateGuidelines}(\mathcal{G}, \mathcal{L})$
  end if
end for
return $\mathcal{H}^\star \leftarrow \mathsf{Consolidator}(\mathcal{H})$

Algorithm 2 GENERATEHEURISTIC: propose a task, calibrate it, and mine a heuristic

Input: environments $\mathcal{E}$; survey $\mathcal{S}$; accepted tasks $\mathcal{T}$; guidelines $\mathcal{G}$; coverage $\mathcal{C}$; hyperparameters N_s, N_f, N_r
Output: $(\tau, h, o, \mathcal{L})$: task, candidate heuristic, outcome, and session history

$(\tau, \xi) \leftarrow \mathsf{Explorer.Explore}(\mathcal{E}, \mathcal{S}, \mathcal{T}, \mathcal{G}, \mathcal{C})$ ▷ ξ is the explorer trajectory
$\mathcal{L} \leftarrow [\,]$
$(h, o) \leftarrow$ SOLVERLOOP$(\mathcal{E}, \tau, N_s, N_f)$ ▷ Algorithm 3
append (τ, ξ, o) to $\mathcal{L}$
if $o = \texttt{accepted}$ **then return** $(\tau, h, o, \mathcal{L})$
for $j = 1, \ldots, N_r$ **do**
  $d \leftarrow \texttt{harder}$ **if** $o = \texttt{too_easy}$ **else** $\texttt{easier}$
  $(\tau, \xi) \leftarrow \mathsf{Explorer.Refine}(\mathcal{E}, \tau, \xi, d, \mathcal{L}, \mathcal{G}, \mathcal{C})$
  $(h, o) \leftarrow$ SOLVERLOOP$(\mathcal{E}, \tau, N_s, N_f)$ ▷ Algorithm 3
  append (τ, ξ, o) to $\mathcal{L}$
  if $o = \texttt{accepted}$ **then return** $(\tau, h, o, \mathcal{L})$
end for
return $(\tau, h, o, \mathcal{L})$

Algorithm 3 SOLVERLOOP: mine and validate a heuristic

Input: environments $\mathcal{E}$; task $\tau = (u, \sigma)$; hyperparameters N_s, N_f
Output: (h, o): heuristic (if accepted) and outcome

$h \leftarrow \varnothing$; $streak \leftarrow 0$; $failures \leftarrow 0$
repeat
  $e \leftarrow \texttt{Reset}(\mathcal{E})$; $\zeta \leftarrow \mathsf{Solver}(e, u, h)$
  $passed \leftarrow \mathsf{Judge}(\zeta, u, \sigma)$
  if $passed$ **then**
    $streak \leftarrow streak + 1$
  else
    $streak \leftarrow 0$; $failures \leftarrow failures + 1$
    $h \leftarrow \mathsf{Extractor.Edit}(u, \zeta, h)$ ▷ edits or extends h; overall failure only
  end if
until $streak = N_s$ **or** $failures = N_f$
if $streak = N_s$ **and** $h = \varnothing$ **then return** $(\varnothing, \texttt{too_easy})$
if $streak = N_s$ **then return** $(h, \texttt{accepted})$
return $(h, \texttt{too_hard})$

A.2 DAEDALUS-CURATED

Algorithm 4 Accumulation from labeled tasks (DAEDALUS-curated)

Input: environments $\mathcal{E}$; labeled tasks $\mathcal{Q}$ with verifier V; hyperparameters N_s, N_f
Output: frozen heuristic bank $\mathcal{H}^\star$

$\mathcal{H} \leftarrow [\,]$
for $\tau \in \mathcal{Q}$ **do**
  $(h, o) \leftarrow$ SOLVERLOOP$(\mathcal{E}, \tau, N_s, N_f)$ ▷ Algorithm 3; Judge replaced by V
  if $o = \texttt{accepted}$ **then append** h **to** $\mathcal{H}$
end for
return $\mathcal{H}^\star \leftarrow \mathsf{Consolidator}(\mathcal{H})$

A.3 MEMORY-AUGMENTED INFERENCE

Algorithm 5 Memory-augmented inference (ALL @ START)

Input: environments $\mathcal{E}$; frozen bank $\mathcal{H}^\star$; instruction u
Output: Solver trajectory ζ

$e \leftarrow \texttt{Reset}(\mathcal{E})$
$\zeta \leftarrow \mathsf{Solver}(e, u, \mathcal{H}^\star)$
return ζ

B LIMITATIONS

DAEDALUS builds a memory bank once, before deployment, from a single environment, and relies on an LLM judge checking self-written success conditions as its only success signal. This design makes memory creation fully autonomous but leaves some limitations open.

Memory lifecycle. We study memory construction as a one-time stage and keep the bank frozen at test time. Continuing to update it from deployment experience, for instance by alternating DAEDALUS sessions with test-time reflection, is left to future work; our preliminary attempts at such iterative schemes did not yield consistent gains.

Environment and verification coverage. DAEDALUS assumes that the explorer can set up and complete tasks alone in a resettable environment. Dual-control settings such as τ^2-bench telecom, where a simulated user must also act on the environment, are not yet supported. Success is also

judged by an LLM against self-written conditions, whose agreement with official verifiers is substantial but imperfect (κ_{bench} of 0.73–0.81). For code-based tasks, an executable verifier agent could replace or complement the judge.

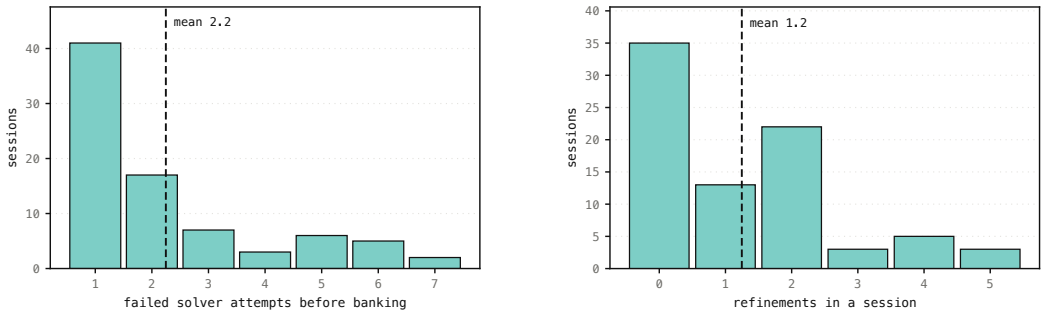
Consolidation at scale. To test whether the 140-session bank is too large for a single consolidation call, we also consolidate its 130 heuristics hierarchically: we split them at random into eight groups and merge these pairwise with the same prompt (Prompt 8) until one bank remains ($8 \to 4 \to 2 \to 1$). Both 140-session banks fall below the 90-session bank overall, yet on the 131 *Other* tasks they nearly match it (Table 7, shaded): most of the gap sits in three small categories, where a few heuristics lost their scope. The single call narrows the 90-session rule to *"enumerate every matching container"*, so the Solver reads a single page of playlists or friends; the hierarchical merge broadens a note on public playlists into locating any playlist with `is_public=True`, which hides private ones.

Table 7: Consolidating the 140-session bank on AppWorld by task category (number of tasks in parentheses). Categories are defined from the task instruction: *Playlists* operates on the user's Spotify playlists; *Venmo friends* reads or edits the Venmo friend list; *Venmo history* aggregates past Venmo payments or requests; *Other* is every remaining task. Two tasks belong to two categories.

		Playlists (11)		Venmo friends (15)		Venmo history (13)		Other (131)		All (168)	
Sessions	Consolidation	MSR	pass^5	MSR	pass^5	MSR	pass^5	MSR	pass^5	MSR	pass^5
no memory		45.5	9.1	42.7	0.0	32.3	0.0	46.0	18.3	44.3	14.9
90	single call	**67.3**	**45.5**	**69.3**	**20.0**	**55.4**	**7.7**	**59.2**	34.4	**60.2**	**32.1**
140	single call	49.1	27.3	36.0	6.7	52.3	0.0	56.2	**36.6**	53.7	30.9
140	hierarchical	41.8	0.0	57.3	13.3	43.1	0.0	56.8	30.5	54.6	25.0

C Memory Generation Settings

Setting the hyperparameters. The generation loop of DAEDALUS is controlled by a failure limit N_f, which caps the number of failed solver attempts at a task, and a number of refinements N_r which sets the number of times the explorer may revise a task that is too easy or too hard. Low limits lose heuristics on the hardest tasks, which are the ones where memory helps most. High limits spend compute on tasks the Solver cannot solve or the Explorer cannot calibrate. We set $N_f = 8$ and $N_r = 5$ after a few preliminary sessions and validate them on the 90-session AppWorld run (Figure 9). Most heuristics are accepted after one or two failures, but a tail of sessions need up to seven, so a smaller N_f would lose heuristics on the hardest feasible tasks. Additionally, most sessions need at most two refinements and only a handful need four or five, which justifies our choice of $N_r = 5$.



(a) Failed solver attempts before $N_s = 3$ consecutive successes.

(b) Task refinements in successful sessions.

Figure 9: Failures and refinements per accepted heuristic over the 90-session AppWorld run.

Impact of the extractor model choice. On the official AppWorld tasks, the benchmark provides the tasks and their verdicts, so the extractor is the only auxiliary agent that affects the memory. We vary its model and reasoning effort, while keeping GPT-5.4-mini as the solver (Table 8). With GPT-5.4-mini at high reasoning effort instead of GPT-5.4, the gain over the no-memory baseline falls from 16.5 to 12.8 MSR points. Disabling reasoning lowers it further to 9.5 points, while pass^5 does not change. Every extractor still improves the solver by a clear margin, so extraction from curated trajectories is robust to the choice of model. The drop from GPT-5.4 to GPT-5.4-mini (3.7 MSR points) is also smaller than on self-generated tasks (9.0 points, Section 5.1), which suggests that the other auxiliary roles account for most of the drop in that setting.

Table 8: Extractor model choice for DAEDALUS-curated on AppWorld.

Extractor	Reasoning effort	# Heuristics	MSR↑	pass^5 ↑
no memory	–	–	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$
GPT-5.4-mini	none	26	$53.8_{\pm 1.4}$	$28.6_{\pm 3.5}$
GPT-5.4-mini	high	38	$57.1_{\pm 1.1}$	$28.6_{\pm 3.5}$
GPT-5.4	high	40	$\mathbf{60.8}_{\pm 0.9}$	$\mathbf{36.3}_{\pm 3.7}$

Counterfactual replay of accepted heuristics. Since the solver is an LLM agent, it is inherently stochastic and an ineffective heuristic could pass the acceptance rule by chance. To quantify to what extent the heuristics we accept raise the solver's success rate, we run the solver three times on each of the 81 accepted AppWorld tasks, with and without their accepted heuristic, and grade it with the LLM Judge. With its heuristic in context, the solver's MSR rises from 66.7% to 77.0% and its pass^3 from 40.7% to 55.6%. We assess these gains with a paired bootstrap, which resamples the 81 tasks with replacement 10,000 times while keeping each task's two conditions together, so that differences in difficulty between tasks do not inflate the variance. The resulting 95% confidence intervals are [2.1, 18.5] points for MSR and [1.2, 28.4] points for pass^3, and both exclude zero.

D MEMORY RETRIEVAL AND INJECTION

We investigate three axes that control how the heuristic bank is used at inference: the retrieval method and the number of retrieved heuristics, the source of the retrieval query, and the injection policy (Figure 10). All experiments use the 65-heuristic consolidated bank generated on AppWorld with GPT-5.4, evaluated on the 168 tasks of `test_normal` over five runs with GPT-5.4-mini as the solver. Unless stated otherwise, retrieval uses the agent's reasoning from the previous turn as query (PRE-GEN), retrieves k=5 heuristics per turn, and keeps them in context with deduplication (CUMULATIVE, DEDUPLICATED); this is the configuration reported in Table 6.

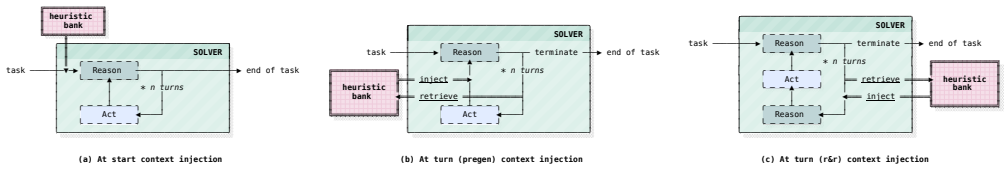


Figure 10: Heuristics injection modes at inference.

The number of retrieved heuristics matters more than the retriever. At k=5, dense retrieval with Qwen3-Embedding-4B, sparse retrieval with BM25, and random selection do not differ significantly (Table 6), indicating that on a consolidated bank, any five heuristics carry similar information regardless of how they are chosen. Increasing k with BM25 improves both MSR and pass^5, which plateau between k=10 and k=20 and rise again at k=30 (Figure 7). Retrieval only approaches whole-bank injection once it surfaces about half of the bank per turn; for a bank of 3.1k tokens, retrieving a subset therefore brings no practical benefit.

Table 9: Retrieval query source on AppWorld. BM25, k=5, same consolidated bank.

Query source	MSR↑	pass^5 ↑	$↓
No memory	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	**3.2**
PRE-GEN	$\mathbf{52.0}_{\pm 1.3}$	$\mathbf{22.0}_{\pm 3.2}$	3.8
R2R	$44.2_{\pm 1.2}$	$11.9_{\pm 2.5}$	14.8

Table 10: Injection modes on AppWorld. Retrieved subsets use Qwen3-Embedding-4B with PRE-GEN queries and k=5. Distinct: mean number of distinct heuristics seen by the solver at episode end. **Bold** marks the best value in each column and underline the second best.

Injection mode	MSR↑	pass^5 ↑	$↓	Distinct
No memory	$44.3_{\pm 1.1}$	$14.9_{\pm 2.8}$	3.2	0
Retrieved subset, k=5 per turn				
TRANSIENT	$48.5_{\pm 0.7}$	$13.1_{\pm 2.6}$	**2.8**	10.8
CUMULATIVE, WITH REPETITION	$50.0_{\pm 1.7}$	$17.3_{\pm 2.9}$	3.1	11.7
CUMULATIVE, DEDUPLICATED	$54.3_{\pm 1.6}$	$24.4_{\pm 3.3}$	3.7	12.1
CUMULATIVE, NO REPLACEMENT	$\underline{56.0}_{\pm 1.2}$	$25.6_{\pm 3.4}$	3.9	46.0
Whole bank				
ALL @ TURN (transient)	$51.5_{\pm 1.2}$	$\underline{28.0}_{\pm 3.5}$	3.5	65
ALL @ START	$\mathbf{60.2}_{\pm 0.9}$	$\mathbf{32.1}_{\pm 3.6}$	3.2	65

Writing a dedicated retrieval query does not pay off. We compare two ways of forming the retrieval query (Table 9). PRE-GEN retrieves on the reasoning the agent produced at the previous turn, whereas R2R spends a second LLM call per turn to write a dedicated retrieval query. With BM25 in both cases, R2R more than triples the cost per run (14.8 vs. 3.8) and falls to the no-memory level on MSR (44.2 vs. 44.3) and below it on pass^5 (11.9 vs. 14.9). Writing a dedicated query does not make retrieval more precise and mostly adds noise, so we use PRE-GEN whenever retrieval is needed.

Heuristics help most when they persist in context. In Table 10, the retriever is fixed and only the way retrieved heuristics enter the context changes. TRANSIENT shows them for the current call and then drops them from the history. The other three policies keep them in the history and differ only in how they treat a heuristic that is retrieved again. WITH REPETITION appends it anyway, DEDUPLICATED skips it, and NO REPLACEMENT removes it from the candidates altogether, so that each turn brings five heuristics the agent has not seen before. Retention is the dominant factor: TRANSIENT is the weakest policy (48.5 MSR), and its pass^5 of 13.1 falls below the no-memory baseline. Repeating heuristics already in context also hurts relative to deduplication (50.0 vs. 54.3 MSR), suggesting that redundant context harms multi-turn decision-making. NO REPLACEMENT exposes the solver to the most distinct heuristics (46.0 on average) and achieves the best retrieved-subset MSR (56.0), yet remains below whole-bank injection at task start (60.2). The same pattern holds for the whole bank: injecting it transiently at every turn (ALL @ TURN) underperforms injecting it once in the system prompt (ALL @ START) on both metrics and at higher cost, consistent with the benefit of persistent guidance.

E EXPERIMENTAL SETUP DETAILS

This section describes the configurations used to produce the main results (Table 1).

Models and shared settings. Every method uses the same pair of LLMs on a given benchmark (Table 11). The solver and the base agent run on the smaller model, and every auxiliary agent runs on the larger model. This includes roles that some original implementations assign to the solver model: the ExpeL reflector, the AutoGuide context module, the ReasoningBank extractor and judge, and the PREPING proposer, validator and curator. We raise them to the larger model so that no method

is limited by a weaker auxiliary model. On τ^2-bench, the user simulator is `gpt-4.1-2025-04-14` for all methods, as per the original benchmark implementation.

Table 11: Models and solver step ceiling for each benchmark (reasoning effort in parentheses).

Benchmark	Solver / base agent	Auxiliary agents	Step ceiling
AppWorld	`gpt-5.4-mini` (default)	`gpt-5.4` (high)	30
τ^2-bench	`gpt-5.4-mini` (default)	`gpt-5.4` (high)	200
AutomationBench	`gpt-5.6-luna` (medium)	`gpt-5.6-terra` (high)	50

DAEDALUS. Table 12 lists the hyperparameters introduced in Algorithms 1–3. DAEDALUS-curated uses the same values on the training tasks, with the benchmark verifier in place of the Judge. On the 90-session AppWorld run, an early experimental filter discarded seven accepted tasks whose expected tool paths near-duplicated banked tasks. We replaced them with seven successful sessions from the scaling experiment (Figure 6), yielding 88 accepted heuristics over the 90 retained sessions (97.8%). This filter was not part of the general method and was not used on other benchmarks. Analyses that use the tasks of this run (Section 5.3, Figure 9, and the counterfactual replay in Appendix C) use only the 81 tasks accepted in the original sessions, without the seven replacements.

Table 12: DAEDALUS hyperparameters.

Hyperparameter	Value
Explorer turn ceiling	40
Concurrent explorers	5
Max task refinements N_r	5
Max failed attempts N_f	8
Consecutive successes to accept N_s	3

τ^2-bench retail environment. In the retail domain, the tool `modify_pending_order_items` assigns the price and options of the last exchanged pair to every item of a multi-item call, so two equally correct solutions that list the same items in a different order reach different database states and receive different grades. We do not modify the environment or the benchmark tasks, which keeps our scores comparable to the original implementation. Instead, the DAEDALUS Explorer rejects any generated task whose reference solution exchanges more than one item in a single call, so that generated tasks are not graded by argument order.

Baselines. We reimplement ExpeL, ERL, AutoGuide, ReasoningBank and ACE on top of our solver harness, with the hyperparameters of Table 13. For ExpeL, the insight list cap follows the authors' released code, and we use $L = 3$ instead of the paper's 4 to 8 because our trajectories are much longer. For AutoGuide, we use $k = 3$, the best value in the paper's own ablation. In ERL, we use both successful and failed attempts to create heuristics. ReasoningBank builds its bank on the training tasks, processed in parallel waves, and the bank is frozen at test time so we do not use test-time scaling (MaTTS). We use `text-embedding-3-large` as the embedding model to retrieve memories. For ACE, we use its offline variant without ground truth: each training task is solved once, sequentially, and a reflector and a curator turn every trajectory into ADD operations on the playbook, with no success signal. We start from an empty playbook, run the reflector and the curator on `gpt-5.4`, and inject the whole playbook at the start of each test task. For PREPING, we run the authors' released code and inject the whole playbook at the start of each test task. On τ^2-bench, its task proposer also sees sampled database records, without which it could not write feasible tasks.

Memory construction budget. DAEDALUS spends more solver rollouts than any baseline, about 6 times more than ExpeL and 14 times more than single-attempt methods (Table 14). Most of this gap is the cost of not having training tasks. Of the 1,225 rollouts, 520 are spent on the explorer's first version of each task, which is close to the 592 rollouts of DAEDALUS-curated on the training tasks, and the remaining 705 are spent on refined versions of tasks that were too easy or too hard. The Solver loop itself therefore costs about as much on generated tasks as on curated ones, and task calibration roughly doubles it. These rollouts use the small Solver model, so they account for a minor share of the generation cost reported in Table 3, which is dominated by the explorer and the judge.

Table 13: Baseline hyperparameters.

Method	Hyperparameter	Value
ExpeL	Max Reflexion retries per task Z	3
	Successes per comparison L	3
	Max success/failure pairs per task	3
	Insight list cap	20
	Few-shot examples k	2
	Retrieval embedder	`all-mpnet-base-v2`
ERL	Heuristics selected k	20
AutoGuide	Max Reflexion retries per task	3
	Max success/failure pairs per task	3
	Guidelines injected k	3
ReasoningBank	Memory items per trajectory	≤ 3
	Experiences retrieved k	1
	Retrieval embedder	`text-embedding-3-large`
	Tasks per wave	5 / 5 / 8
ACE	Rollouts per training task	1
	Tasks per wave	1
	Curator operations	ADD only
	Initial playbook	empty (8 sections)
	Playbook token budget	80k
	Final playbook bullets	240 / 164 / 103
PREPING	Cycles $\times$ tasks per cycle	10×10
	Feasibility score to update the playbook	5
	Completion score to count as success	≥ 4

Environment access during generation. DAEDALUS never interacts with the environment states of test tasks. On AppWorld, each generation session runs in the initial world of one training task, whose instruction and verifier are not used, and generation excludes Amazon and Gmail, which appear only in the challenge split (Trivedi et al., 2024). The Explorer therefore sees the same environment states as methods using training tasks. On AutomationBench, generation uses only the worlds of the 30 training tasks. On τ^2-bench, all tasks share a single database, which methods using training tasks also access.

Table 14: Solver rollouts spent building memory on AppWorld. Each DAEDALUS session counts as one training task. Explorer interactions are not included.

Method	Solver rollouts	Per task	MSR
ERL	90	1.0	58.6
ReasoningBank	90	1.0	48.0
ACE	90	1.0	60.5
ExpeL	200	2.2	59.0
AutoGuide	200	2.2	44.0
DAEDALUS-curated	592	6.6	60.8
DAEDALUS	1,225	13.6	60.2

F GENERATION EXAMPLES ON APPWORLD

F.1 GENERATED TASK AND ACCEPTED HEURISTIC

Table 15 shows one task generated by the Explorer on AppWorld, the success conditions the Judge checks, and the heuristic accepted after the Solver loop. Section F.2 details a full generation session.

Table 15: AppWorld example

Task example	Please text everyone from my latest friends dinner note to let them know how much they owe me for that dinner.
Success conditions	- Compared with the starting state, every person listed in the latest friends dinner note has a new outgoing text message from Kristin in that contact's message thread. - Each new text correctly states the amount assigned to that person in the note and makes clear it is about that dinner.
Accepted heuristic	- When using `simple_note`, the available note-reading methods are `search_notes` and `show_note`; there is no `list_notes`, so search by query first and then open the chosen `note_id`. - When API-doc search results are suppressed, print `dir(apis.<app>)` to discover exact method names and proceed from those concrete names instead of guessing endpoints. - When a cross-app workflow needs another app after notes, get and use that app's own access token; tokens are app-specific and a token from one app will 401 on another app's endpoints. - When a login with the main profile email fails for the phone app, try authenticating with the user's phone number as the username, because that app may use phone-number-based login rather than email. - When a search query returns noisy matches, do not trust the query alone; inspect the returned titles and timestamps and open the candidate note to confirm it is the latest relevant one before acting on its contents.

F.2 A COMPLETE GENERATION SESSION

Session 78 begins.

After 12 turns of exploration, the **EXPLORER** proposes a task T1: "*Please assign me to every unfinished unassigned task in any of my shared Todoist projects, not the subtasks.*"

The **SOLVER** tries task T1: it succeeds 3 times in a row ✅. The task is too easy: it teaches the memory bank nothing, because the **EXTRACTOR** only writes on failure.

The **EXPLORER** enters refinement #1: it must make the task harder. The **EXPLORER** understands that using '*unassigned*' made the task too easy: the **SOLVER** skips every task that already has an owner, so the hard part (reading who is assigned to what) never happens. After 3 more exploration turns, it proposes task T2: "*Please assign me to every unfinished task in any of my shared Todoist projects, not the subtasks.*"

The **SOLVER** tries task T2: it spends 21 turns guessing the assignment API's signature, finally manages to do one tool call, submits, and fails ❌.

The **EXTRACTOR** writes heuristics from the failed trajectory:

> * *When working with Todoist in this environment, inspect `dir(apis.todoist)` first and use the exact exposed method names such as `show_projects`, `show_tasks`, and `assign_or_unassign_task` instead of guessing API names from generic docs.*
> * *When filtering Todoist tasks for unfinished top-level items, use the returned task fields `is_completed == False` and `num_sub_tasks == 0`, after checking the actual task shape from `show_tasks`.*
> * *When assigning a Todoist task, call `assign_or_unassign_task(task_id=..., assignee_email=..., access_token=...)`; do not pass `project_id` or an assignee object.*

The **SOLVER** attempts task T2 with these heuristics in context, and fails ❌. Its trajectory shows that it opened one project (#155), hand-copied a literal list of 12 task ids out of the printed output, assigned those, re-read the same 12 to confirm, and called `complete_task()`. It never enumerated all the projects, so it missed many tasks.

The **EXTRACTOR** revises the heuristics:

> * *When working with Todoist in this environment, inspect `dir(apis.todoist)` [...]*
> + * *When identifying shared Todoist projects, inspect project records from `show_projects` and filter by collaboration fields such as multiple `collaborators` or `pending_invites`, rather than assuming from the name or using all projects.*
> - * *When filtering Todoist tasks for unfinished top-level items, use the returned task fields `is_completed == False` and `num_sub_tasks == 0`, after checking the actual task shape.*
> * *When assigning a Todoist task, call `assign_or_unassign_task(task_id=..., assignee_email=..., access_token=...)`; do not pass `project_id` or an assignee object.*
> + * *When a Todoist action appears to succeed, do not call `apis.supervisor.complete_task()` until you have checked every relevant shared project and verified all qualifying tasks were handled, because partial success can still fail the overall task.*

The **SOLVER** re-attempts task T2 with the revised heuristic: success ✅. Its trajectory shows that the **SOLVER** now prints the full `show_projects` record, which exposes the `collaborators` field; it names both shared projects #155 and #129, and assigns 27 tasks instead of 12. The two next attempts are successful too ✅. Three in a row: the heuristic is banked.

Session 78 is successful. Session 79 can now begin.

Figure 11: Session 78 from the DAEDALUS generation process on AppWorld (partial).

F.3 SURVEYOR OUTPUT

AppWorld's Survey Output

Axis — The useful spread here is by app/domain involvement, not by raw endpoint count: Spotify, Todoist, Splitwise, Venmo, Phone, File System, Simple Note, plus the tiny meta areas api_docs and supervisor. That partition matches the real records and errands I actually saw: Spotify has 324 searchable songs over 17 pages plus playlists, queue state and premium receipts; Todoist has shared projects/sections/tasks/subtasks/comments/attachments; Splitwise has groups, expenses, payments and balances; Venmo has transactions, requests, balance and comments; Phone has 133 texts over 7 pages plus contacts and alarms; File System and Simple Note are much narrower CRUD helpers; api_docs and supervisor are mostly scaffolding. Because many good tasks span two apps, treat these as involvement buckets rather than forcing every task into one exclusive owner.

Distribution — Weighted, with most of the run in the four rich end-user domains and only a small cap on meta/shallow work. Spotify — 22% — broad, non-interchangeable music work: library, playlists, queue/player, likes/downloads, and receipt export to file_system all behaved differently. Todoist — 18% — real planning/collaboration work is dense even with only 4 projects listed; `delete_section` removed task `6334` and subtask `9636`, and invites/comments/attachments add distinct task shapes. Splitwise — 17% — fewer surfaces than Spotify but high-value reconciliation work: expense `5040` and payment `6015` changed balances immediately, and `delete_group 274` cascaded to associated expenses/payments.
[...]

Guidance for the task generator — A well-covered run should lean hard into Spotify, Todoist, Splitwise and Venmo, with Phone used regularly and File System often as a helper in cross-app tasks, not just as isolated mkdir/cat boilerplate.
[...]

Pitfalls — The easiest over-coverage trap is shallow CRUD in Simple Note, File System, api_docs and supervisor: yes, they work, but too many "create a note", "make a folder", or "show the active task" requests would miss where the environment's real complexity lives. The easiest under-coverage trap is Splitwise and Todoist, whose value exceeds their surface count: balances changed after every expense/payment write, `delete_group` soft-deleted related records, and `delete_section` erased its tasks and subtasks.
[...]

G PROMPTS

Each prompt combines a template shared by all benchmarks (Appendix G.1) with benchmark-specific blocks, such as a description of the environment and the protocol for acting in it. We reproduce the shared templates, the AppWorld solver blocks, and the two benchmark-specific Extractor prompts (Appendix G.2); all prompts are available in full in the GitHub repository. Prompts marked `(partial)` are truncated, with `[...]` marking an elided passage. Values filled in at run time appear as *<placeholders>*, and [Identical to Prompt N] marks text shared with another prompt. Boxes are colored by role: task generation (**blue**), judging (**red**), memory extraction and consolidation (**green**), and solving (**yellow**).

G.1 DAEDALUS PROMPTS

Prompt 1: Surveyor (shared template, partial)

You are an environment analyst. A task-generation system will soon invent many tasks in this environment (a solver attempts them, so reusable know-how can be mined from the attempts). Your job is not to design tasks. Your job is to work out, from the environment itself, what a good spread of those tasks looks like — the coverage goal — and to justify it from what you observe.

The question to answer

Over a whole generation run the tasks should not pile onto one comfortable corner. But "spread out" only means something once you choose the axis to spread along and the distribution over it. Both are yours to decide:

- The axis might be a sub-area or module, a group of related operations, a capability or operation type, a data domain, or a task shape (a question answered from the data, a sweep over everything that matches, a stateful change, a multi-area reconciliation), or a combination. Choose whatever genuinely partitions the useful work here, and say why even coverage of the obvious partition would or would not be right.

- The distribution can be even or weighted. **Do not weight by how many operations an area exposes.** Endpoint count measures how much API was written, not how much real work lives there: an area with sixty operations may be sixty variations on one shallow idea, while an area with twelve may carry most of what a person actually does here. Weight instead by how much of a real person's genuine use of this environment falls in each area, and by how much distinct, non-interchangeable work each supports. Give a concrete emphasis and a one-line reason per unit, and if you find yourself ranking areas in the same order as their operation counts, stop and re-justify it.

[. . .]

You may not deliver the report until you can answer all five of these for EVERY area, from something you actually ran — not from its documentation, and not by analogy with another area:

1. What does a real record here contain, and which of its fields connect it to another area?
2. What is past the first page of its main listing — how many pages are there, and does the listing behave differently beyond page one?
3. What happens when you pass a plausible but wrong identifier, and what happens when you pass a stale one from an earlier call?
4. What does one of its write operations do to *related* records, checked by reading them back afterwards?
5. What does it refuse to do, and with what error — a policy limit, an eligibility rule, or an already-in-that-state no-op?

[. . .]

Output, part 1: the coverage goal

Four short subsections, in this order, each under its own `**heading**` and written as a few plain sentences:

Axis — the axis (or combination) you chose, and why it is the right partition here, with the evidence.
Distribution — even or weighted, and the emphasis for each unit along the axis with a short reason, grounded in realistic use rather than endpoint count. This is the one place where a compact list helps: one line per unit, `unit − share − reason`.
Guidance for the task generator — what to emphasize, what to cap, how to vary the kind of request, and what a well-covered run looks like.
Pitfalls — what is easy to over-cover (shallow or boilerplate areas), what is easy to under-cover (areas whose real value exceeds their surface count), and the traps a naive generator would miss (exact ids, pagination, completeness, cross-area steps).

[. . .]

Output, part 2: the tag manifest

The report on its own cannot be measured, so also define a closed set of tags spanning the axis you chose. Each tag is a short, reusable label a designer assigns to the task they just built. Each carries:

- `tag`: a short, stable, '#'-prefixed label, unique in the set;
- `definition`: one line stating when a designer should assign it, concrete enough that two designers would agree;
- `target_share`: a number in (0, 1], the fraction of banked tasks that should carry the tag.

Three rules decide whether a tag is worth defining, and they matter more than how many you write:

A tag must be able to come out false. If nearly every task you would want here carries it, it measures nothing and steers nothing — a designer sees it satisfied and moves on. Anything you would give a target much above half is a baseline requirement, not a coverage axis: say it in **Guidance** instead and leave it out of the manifest. Prefer tags whose target share sits in the range where being under it is a real, visible gap.

[. . .]

<benchmark-specific deliverable-partition rule, if any>

[. . .]

<benchmark-specific survey protocol>

Prompt 2: Explorer (shared template, partial)

You are a task designer exploring a live environment. Invent ONE task that is genuinely solvable HERE, confirm it is solvable, and hand off a clean specification for a separate solver agent to attempt later.

<benchmark-specific task guidelines>

What makes a good task

- Grounded: it references data that actually exists here (real records, entities, values you discovered) — never invented ones. Inspect the data first to find the specifics.

[. . .]

- Natural: it reads like a real request a person would actually make — a plain, purposeful errand or question, not a puzzle. Difficulty must come from the genuine effort of the request and from

navigating real policy/eligibility rules — NOT from contrived constraints a person would never state. Strictly avoid invented metrics, multi-level tie-breaker chains, embedded lists, and any instruction a human would never spell out.
- Under-specified on purpose: leave OUT every value the solver could discover for itself. Naming a value you had to go and find deletes the work you meant to set. State a literal only when the person supplies it from outside the environment.
- Hard enough, in the right way: calibrate against a CAPABLE solver — if a competent agent would very likely finish it on the first try, it is too easy, and an easy task is wasted effort for the whole run. But get that difficulty from SCOPE, not from size: from what the request leaves the solver to decide (which records qualify, how far back to look, whether there is more than one page, what to do when several match or none do), and from real policy or eligibility limits. Do NOT reach for difficulty by chaining more steps, touching more parts of the environment, or demanding more side effects — that makes a task longer, not harder. But a SHORT request is not a SMALL job: keep the wording brief and the work substantial — one clause covering many records or several pages. A task the solver finishes in a handful of calls is too small however naturally it reads.

[...]

Tasks already generated (do something DIFFERENT):
- *<previously accepted task 1>*
- *<previously accepted task 2>*
- ...

Coverage goal (from the one-time environment survey)
A read-only survey of this environment decided how the whole run's tasks should be spread. Treat it as the coverage plan you are executing:
<coverage-goal report produced by the Surveyor>

Coverage so far — steer toward the UNDER tags
<live tally of coverage tags over the banked tasks, marked UNDER/OK/OVER>

Guidelines from past generations
Follow these closely — they encode what made previous tasks too easy or too hard:
- *<evolved guideline 1>*
- *<evolved guideline 2>*
- ...

[...]

The specification you produce
However you emit it (see the protocol below), your specification has three parts:
- the task: a short, natural-language request a person would actually make. Self-contained in the sense that nothing OUTSIDE the environment is missing — not in the sense of listing what is inside it. Before you emit it, reread it once and delete every id, address, path, amount, date, count and record name that the solver could have found by looking; refer to those things the way the person would. Then check it states a goal rather than a sequence of steps;
- the minimal solution: the ordered tool calls that would solve it (not your exploration trail);
- success conditions: outcome-based, path-agnostic checks on the END STATE, each expressed as a change from the starting state. Never reference which tools were used or how. Only require what the task actually determines — do not bake in incidental details (exact free-text wording, ids, timestamps) the solver could not have known, unless the task itself dictates them; allow any reasonable phrasing for free text.

<benchmark-specific environment block>

<benchmark-specific explorer protocol>

Prompt 3: Explorer task-design doctrine (shared, partial)

Write a request, not a specification. One to three short sentences. A person asking for something does not enumerate the steps, does not spell out the edge cases, and does not paste in record ids. If your instruction has grown into a paragraph with semicolons and a step-by-step plan, it has stopped being a request.

State the goal; never the procedure. Say what *<subject>* wants to be true afterwards, or what they want to know. Do not sequence the work with "then", "after that", "next", "finally". Do not say which *<route_noun>* or which call to use for each part. Working out the route IS the task — if you hand over the route, the *<actor>* only has to type.

Short to state is not small to do. Keep the words brief and the work substantial: *<example>*. Shorten the wording, never the amount of *<swept_noun>* that has to be *<verb>*. A request the *<actor>* finishes in a handful of calls is too small however well it reads.

Say scope the way people do — a superlative (*<superlative>*), a universal (*<universal>*), or a relative time frame (*<timeframe>*). Each is short to write and expensive to get right, so treat them as your main source of difficulty, not as tricks to avoid.
[. . .]

Prompt 4: Explorer, refinement of an off-target task (shared template)

A task you designed earlier turned out ***<too_easy | too_hard>*** when a solver attempted it. Revise it so it lands closer to the right difficulty — challenging but solvable — while keeping it the SAME task in spirit (same goal and style). Make it slightly *<harder | easier>* with a small, targeted change, not a redesign; keep it grounded and natural, and re-confirm it is solvable before you emit it (produce the same specification under the same rules as your original design).

<benchmark-specific task guidelines>

Those rules bind the revision too. In particular, when a task came back **too_hard** the fix is NOT to name the records, spell out the steps, or narrow it to a listed subset — that deletes the work instead of easing it. Make it easier by reducing how much has to be swept or reconciled, or by removing one requirement, while the request stays short and still makes the solver find things for itself.

The task as it stands:
<current task specification>

Revisions already tried this session (do NOT repeat these):
- [*<too_easy | too_hard>*] *<earlier variant>*

Your earlier exploration (reuse what you already learned — you need not rediscover it):
<transcript of the Explorer's earlier exploration>

<benchmark-specific refinement protocol>

Prompt 5: Explorer guideline update (partial)

You maintain a list of guidelines for a task designer. The designer invents tasks in a live environment that are then attempted by a separate solver agent. A task is only useful if it is **challenging but solvable** — neither trivial (solved on the first attempt) nor impossible (never solved) — AND it must read like a natural request a real person would make, never a contrived puzzle. Your guidelines steer the designer toward that frontier WITHOUT sacrificing naturalness. Difficulty should come from genuine work (cross-referencing real data across tools, applications, or data areas, several real steps), never from artificial rules — so do NOT write guidelines that push invented metrics, multi-level tie-breakers, embedded lists, or other things a human would never say. Concrete, environment-specific guidelines are welcome when the evidence supports them.
[. . .]

Current guidelines:
- *<evolved guideline 1>*
- *<evolved guideline 2>*
- ...

All tasks so far that were TOO EASY (solved on the first attempt):
- *<too-easy task>*
- ...

All tasks so far that were TOO HARD (never solved):
- *<too-hard task>*
- ...

This session's trajectory — the designer proposed a task, the solver attempted it, and (if it was off-target) the designer revised it up to several times. Each entry shows the task variant, how it turned out (too_easy / too_hard / banked = good difficulty), and the solver's trace. Study the PROGRESSION to learn what kind of change moved the difficulty in the right direction (or failed to):

— variant 1: *<too_easy | too_hard | banked>* —
TASK: *<task variant>*
SOLVER TRACE:
<solver trace>

Update the guideline list using the evidence above — especially how the difficulty changed across the variants. You may ADD a new guideline, MODIFY an existing one, or DELETE one that the evidence now contradicts. Keep guidelines that still hold.
[. . .]

Maintain a COMPACT, curated playbook of DISTINCT, general design principles — it is the only difficulty signal the task designer sees, so keep it sharp. This is a curated set, NOT a running log: every guideline must earn its place, and the list must not grow without bound. Cover what makes a task too easy here, what makes one too hard, and the concrete design levers in between that raise difficulty *naturally*. Never suggest artificial difficulty. Each guideline is a GENERAL principle — concrete and specific about task *design* and *difficulty*, never referencing specific entities or values — not a session-specific observation.
[...]
Output ONLY the consolidated list — one guideline per line, most important first, no numbering, no prose, no markdown.

Prompt 6: LLM judge (partial)

SYSTEM
You verify whether an agent achieved a task's intended outcomes. Judge the end effect, not exhaustive proof: a successful tool call (optionally confirmed by a read) is enough evidence its effect happened. Fail only on clear evidence a required outcome was missed or wrong; give competent solutions the benefit of the doubt and don't penalize incidental details you can't see.
USER
You are evaluating whether an agent completed a task. Judge by the EFFECTS visible in its trajectory — not by the agent's own claims, and not by demanding exhaustive proof of every detail.
TASK:
<task instruction>
The task counts as COMPLETE only if all of these intended outcomes hold at the end:
1. *<success condition 1>*
2. *<success condition 2>*
3. ...
AGENT TRAJECTORY (the actions it took — code, tool calls, and/or messages — and the observed results):
<solver trajectory>
How to judge, outcome by outcome:
[...]
- Trust effects, not narration. A tool call that returns a success message (optionally confirmed by a follow-up read) is sufficient evidence its effect happened — you do NOT need every field re-printed. Conversely, the agent saying "done" or otherwise signaling completion does NOT by itself prove anything.
- An outcome is UNMET if the trajectory shows it was skipped, applied to the wrong target or value, left an unresolved error, or only partially done.

[...]
- An outcome covering a GROUP ("every", "each", "all the ... that ...") is met only on evidence about the whole group, not about the records the agent happened to handle. A listing can come back partial and look complete, so N actions on one listing's N results say nothing about the group's size. If the trajectory gives you no way to establish that size, that outcome is UNMET — the benefit of the doubt above covers ambiguous detail, never completeness.

The task SUCCEEDS only if every intended outcome is met; otherwise it FAILS.
Output STRICTLY one JSON object on a single line, no prose, no code fences:
`{"success": true or false, "reason": "<one short sentence naming the deciding factor>"}`

Prompt 7: Extractor (shared; used as is on AppWorld; partial)

SYSTEM
You help an agent build reusable memory from task attempts. Given a trajectory and outcome, produce or refine a single memory item that captures actionable lessons for future attempts. Output ONLY the memory text — no headers, no JSON, no fences.
USER
You maintain a running memory for a specific task the agent is retrying. The memory is a bullet list of discrete, actionable lessons that is injected into the agent's prompt on its next attempt. The agent has just made an attempt; produce the updated list.
Task: *<task instruction>*

Attempt: #*<attempt index>* (failure limit: *<max failures>*)
Outcome: *<success | failure>*
Current memory (from previous attempts):
<current heuristic, if any>
Agent trajectory:
<solver trajectory>

How to update the list

If a memory already exists, rewrite it rather than appending: copy correct bullets unchanged, edit ones that need refining, drop ones this attempt proved wrong, and add bullets for what was just learned. Prefer a few sharp bullets over many vague ones.

[. . .]

Look across all the failure dimensions, not only execution errors:

- Task comprehension: if the agent acted without reading all of the task's context (e.g. replied before reading the message that prompted the request, or acted on everything instead of the filtered subset the task names), add a bullet on what to read first and why.
- Answer submission: if the task wants a factual answer (a count, title, name, yes/no) and the agent worked it out but never returned it in the form the environment expects, add a bullet on returning the answer correctly.
- Efficiency: if the agent did more work than needed — e.g. retrieved far more than necessary and filtered afterwards when the environment can filter or query directly — add a bullet recommending the targeted call.

[. . .]

Output

Write each bullet as a single line beginning with `- `:

- When <situation>, <do this>[, because <the reason>]. Keep every bullet self-contained and generalizable — no task-specific identifiers (user ids, emails, values) and nothing that must be derived while solving (e.g. "the count is X"). Output only the bullet list, starting directly with `- `. No headers, JSON, fences, or commentary.

Prompt 8: Consolidation prompt (partial)

You are an expert at distilling an AI agent's scattered experience into a compact, high-signal playbook. The agent solves tasks in an interactive environment by issuing tool / API calls (in some environments, by writing and executing code that calls them). Below are *<N>* reflection notes the agent wrote after solving individual tasks. They are extremely REDUNDANT (the same few lessons restated many times) and polluted with task-specific detail ("for this task, first look up record X, then . . . "). Your job: turn them into a clean playbook of ATOMIC, CONCRETE, GENERALIZABLE heuristics. Rules for the output heuristics:

1. ATOMIC — each heuristic states exactly ONE lesson, in a sentence or two. No multi-lesson bullets.
2. DEDUPLICATE — when notes clearly restate the SAME lesson, emit it once in its sharpest form; if you are unsure whether two notes make the same point, keep both.
3. CONCRETE — preserve exact, fully-qualified tool / API / function names and parameter names EXACTLY as the notes give them, in whatever call syntax the environment uses. Concreteness is the whole point; never abstract a real call into vague advice.
4. GENERALIZABLE — state reusable mechanisms, not recipes welded to one task. DROP every "for this task / for this workflow" instruction and anything that only makes sense for one specific task's data.

[. . .]

Return STRICT JSON only (no markdown fence), an object:
`{"heuristics": [{"topic": "<short topic tag for the sub-area, e.g. general | auth | discovery | output | completion | or an environment-specific tool/area name>", "text": "<the atomic heuristic>"}, ...]}` Output the smallest set of heuristics that still fully covers the material — let the notes decide the count; never aim for a target number. Two goals in tension:

1. COVERAGE — nothing useful is lost, so every distinct reusable lesson the notes contain is represented, spanning both general mechanisms that apply to every task and environment-specific facts about the particular tools or areas the notes cover;
2. MINIMALITY — no redundancy, each distinct lesson stated exactly once in its sharpest form, merging overlapping notes rather than listing variants (do not pad the list).

Here are the notes:
<numbered list of the N retained heuristics>

Prompt 9: Solver (shared template)

You are an autonomous agent that completes a user's task in a live environment.

Work carefully and deliberately:

- Read the task and identify every condition, filter, and constraint it states, and make sure your solution satisfies all of them.
- Inspect the real data before you act — never assume a value or an identifier; look it up in the environment.
- Proceed one step at a time: after each action, check the result before deciding the next step.
- Handle errors gracefully — if an action fails, read the error and adjust rather than repeating it.
- Verify that the intended outcome actually holds before you finish, and finish only once the task is genuinely complete.

<benchmark-specific environment block>

<benchmark-specific solver protocol>

Learned guidelines from previous experience. These guidelines are advisory and may not apply to the current task. Use only those that are consistent with the task, the environment policy, the available tools, and the data you observe.

1. *<heuristic 1>*
2. *<heuristic 2>*
3. ...

G.2 BENCHMARK-SPECIFIC PROMPTS

The original AppWorld solver prompt (Trivedi et al., 2024) contains extensive human-written heuristics for the environment, so we replace it with a simplified version (Prompt 10, inserted into the shared solver template of Prompt 9) to evaluate all methods with less hand-engineered guidance.

Prompt 10: AppWorld solver used in this work (blocks of Prompt 9)

ENVIRONMENT BLOCK

What you already know about this environment

- Most app APIs are protected. Get credentials with `apis.supervisor.show_account_passwords()` (a list of `{account_name, password}`) and pick the entry matching the app you need.
- Log in through the app's own login method (e.g. `apis.<app>.login(username=..., password=...)`). The username is usually the main user's profile email, not their display name.
- Do not assume a login session persists: capture the returned `access_token` and pass it explicitly (`access_token=...`) to every later protected call for that app.
- Before filtering or aggregating records, inspect one returned item's fields so you filter on the real field name rather than guessing.

PROTOCOL

You act by writing and executing Python code in a persistent, interactive environment (variables persist between steps). Each turn, respond in exactly this format:

Thought: <your reasoning about what to do next>

```python
<your code>
```

- Call application APIs directly as `apis.<app_name>.<method_name>(...)`.
- Discover APIs with `apis.api_docs.search_api_docs(query="...")` (free-form text) or `dir(apis.<app_name>)`, then `apis.api_docs.show_api_doc(app_name="...", api_name="...")` for the full documentation.
- When the task is complete, call `apis.supervisor.complete_task()`. If and only if the task asks for a factual answer (a number, name, title, yes/no, ...), pass it as `apis.supervisor.complete_task(answer=<your_answer>)` where the answer is a concise string or number. If the task only requires side effects (sending a message, saving a file, following an artist, ...), call it with no arguments.
- Make `complete_task(...)` the ONLY statement in its code block, and submit it only after an earlier turn has verified the result (re-read the records you changed, re-derive any answer from returned data). Do not compute or fix anything in the same block as `complete_task`.

The applications available to you (these are the ONLY ones — treat any other app as nonexistent):

<JSON list of app names and descriptions>

The main user you are helping is: *<supervisor's name, email and phone number>*. Authenticate as needed.

Prompt 11: τ^2-bench Extractor (partial)

SYSTEM

[Identical to Prompt 7]

USER

You maintain a running memory for a specific customer-service task the agent is retrying. The memory is a bullet list of discrete, actionable lessons that is injected into the agent's prompt on its next attempt. The agent has just made an attempt; produce the updated list.

[Identical to Prompt 7]

Agent trajectory (conversation + tool calls):
<solver trajectory>

[Identical to Prompt 7]

[. . .]

Look across all the failure dimensions, not only tool errors:

- Policy compliance: if the agent broke a policy rule (skipped a required eligibility check, acted without the mandatory explicit confirmation before a modify/cancel/refund, or exceeded what policy allows), add a bullet stating the rule and when it applies.
- Information gathering: if the agent acted before authenticating the user or before looking up the records the action depends on, add a bullet on what to establish first and why.
- Tool arguments: if a call failed or silently did the wrong thing from a wrong or missing argument (wrong id, wrong enum value, a list where one item was meant), add a bullet with the correct usage.
- Communication: if the task needed the agent to tell the user something (a total, a status, an eligibility answer) and it never did, add a bullet on reporting results before ending.
- Ending: if the agent stopped or transferred early, or kept going after the request was complete, add a bullet on when the task is actually done.

Output

Write each bullet as a single line beginning with - :

- When <situation>, <do this>[, because <the reason>]. Keep every bullet self-contained and generalizable — no task-specific identifiers (user ids, order numbers, names, addresses) and nothing that must be derived while solving (e.g. a specific order total). Output only the bullet list, starting directly with - . No headers, JSON, fences, or commentary.

Prompt 12: AutomationBench Extractor (partial)

SYSTEM

[Identical to Prompt 7]

USER

You maintain a running memory for a specific business-operations task the agent is retrying. The task runs against simulated SaaS applications (mail, chat, spreadsheets, calendars, trackers, knowledge bases, CRM, e-signature) reached through a single HTTP tool. The memory is a bullet list of discrete, actionable lessons that is injected into the agent's prompt on its next attempt. The agent has just made an attempt; produce the updated list.

[Identical to Prompt 7]

What makes a bullet worth writing

The agent will next meet a DIFFERENT task, in a DIFFERENT world, wired to a DIFFERENT set of applications. A bullet earns its place only if it would still change what the agent does there.

Write decision rules over the task's own data — which records to read before acting, which ones qualify, which source wins when two disagree, what must be true before a write, what to check after one. These recur across tasks because the reasoning recurs.

Do not write a recipe for one endpoint. A bullet whose situation clause is "to create a ticket in system X" or "to lock a page in system Y" fires only when the next task happens to use that call, and the agent can already discover the call itself. The exception is a mechanical trap that bites across services — argument encoding, placeholder substitution in a discovered URL, what a response does and does not prove — which is worth one bullet stated once, in general terms.

Look across all the failure dimensions, not only the calls that returned an error:

- Evidence gathering: if the agent selected, ranked or excluded records before reading every source that governs the choice (all tabs of a worksheet, the full thread rather than its first message, the channel history rather than the newest post), add a bullet on what to read first and why.
- Qualification: if the agent applied the task's filters, exclusions, holds or date scopes loosely, or applied a requirement to records that never declared it, add a bullet stating how to decide membership.
- Authority: if the agent treated the wrong source as decisive — an older notice over a newer one, an unofficial mention over the authorized sender, a request field over the registered record — add a bullet naming which source governs.

[. . .]

- Write verification: if the agent reported completion on the strength of a call that returned without error, add a bullet on what to re-read to prove the effect landed.

[. . .]

Output

Write each bullet as a single line beginning with - :

- When <situation>, <do this>[, because <the reason>]. State the situation by the shape of the task, not by the application it happened to use. Keep every bullet self-contained and generalizable — no task-specific identifiers (record ids, addresses, spreadsheet or channel names, values) and nothing that must be derived while solving (e.g. "the count is X"). Output only the bullet list, starting directly with - . No headers, JSON, fences, or commentary.