Learning and Reusing Policy Decompositions for Hierarchical Generalized Planning with LLM Agents

Shirin Sohrabi, Haritha Ananthakrishnan, Harsha Kokel, Kavitha Srinivas, Michael Katz

IBM

Abstract

We present a dynamic policy-learning approach that combines generalized planning and hierarchical task decomposition for LLM-based agents. Our method, Hierarchical Component Learning for Generalized Policies (HCL-GP), learns parameterized policies that generalize across task instances and automatically extracts reusable policy components from successful executions, organizing them into a skill library for compositional policy generation. We address three challenges: (1) learning skills through automated decomposition, (2) generalizing skills to maximize reuse, and (3) efficient retrieval via semantic search.

Evaluated on the AppWorld benchmark, our approach achieves 98.2% accuracy on normal tasks and 97.8% on challenge tasks with unseen applications, improving 15.8 points over static synthesis on challenging scenarios. For open-source models, dynamic reuse enables 62.5% success versus near-zero without reuse. This demonstrates that classical planning concepts can be effectively integrated with LLM agents for improved accuracy and efficiency.

Introduction

Interactive coding agents have become increasingly capable of solving realistic tasks by generating executable code that queries APIs, manipulates state, and iteratively repairs failures. Yet many systems treat each task as an isolated generation problem. In environments with repeated structure, this is inefficient: agents repeatedly rediscover solution patterns, spend inference budget on near-duplicate problems, and fail to accumulate reusable procedural knowledge.

We study dynamic policy learning in structured sequential decision-making settings where environments contain multiple domains, each consisting of related task instances that share latent structure but differ in parameters. The objective is to produce generalized, reusable policies that instantiate across instances and adapt efficiently to new domains.

Our approach draws on two AI planning paradigms. First, generalized planning (Bonet et al. 2017; Hodel 2024; Silver et al. 2024; Stein et al. 2026) produces parameterized policies that solve classes of problems by abstracting over instance-specific details. Second, Hierarchical Task Network (HTN) planning (Erol, Hendler, and Nau 1994; Hogg, Muñoz-Avila, and Kuter 2008; Georgievski and Aiello 2015) decomposes complex tasks into reusable subtasks. We integrate these principles with LLM-based agents through a dynamic policy-learning framework that learns parameterized policies and automatically extracts reusable policy components from successful executions. These components are evaluated, generalized, organized into a skill library, and retrieved via semantic search. Policy generation proceeds compositionally by combining retrieved skills with task-specific parameters. In what follows, we refer to policy components outside of the policy as skills.

We instantiate this framework on AppWorld (Trivedi et al. 2024), a benchmark where agents interact with multiple applications (Gmail, Spotify, Phone, etc.) through APIs. Each scenario consists of multiple related tasks sharing structure but varying in parameters, making AppWorld a natural testbed for generalized and compositional policies.

Our contributions are: (1) A dynamic policy-learning framework integrating generalized planning and hierarchical decomposition for reusable, parameterized policies. (2) Methods for automatically extracting, evaluating, and reusing policy components from successful executions. (3) An empirical evaluation on AppWorld demonstrating improved efficiency and favorable cost-accuracy trade-offs, particularly on challenging domains. These results show that planning concepts such as generalized policies and decomposition remain effective tools for structuring LLM-based agents in realistic decision-making environments.

Related Work

Our work is grounded in generalized planning and hierarchical task decomposition. Generalized planning synthesizes policies that solve classes of problems by abstracting instance-specific details (Jiménez, Segovia-Aguas, and Jonsson 2019). Recent work uses LLMs to generate such policies in code (Silver et al. 2024; Stein et al. 2026). We learn parameterized policies and reusable skills that generalize dynamically across instances. Unlike traditional approaches, we do not assume symbolic domain representations; generalization emerges from execution feedback and iterative synthesis.

Hierarchical Task Network (HTN) planning decomposes tasks into reusable subtasks (Erol, Hendler, and Nau 1994; Nau et al. 2003; Ghallab, Nau, and Traverso 2004). HTN

learning aims to automatically identify reusable substructures (Hogg, Muñoz-Avila, and Kuter 2008; Georgievski and Aiello 2015). We adopt this insight but learn hierarchical structure implicitly from LLM-generated executions rather than symbolic plans, and discover skills dynamically rather than enforcing fixed task networks.

Learning reusable skills has been studied in learning from demonstrations (Argall et al. 2009) and hierarchical reinforcement learning (Sutton, Precup, and Singh 1999; Hutsebaut-Buysse, Mets, and Latré 2022). These approaches typically operate in low-level control settings with fixed state-action spaces. We differ by learning reusable skills as executable code fragments that directly implement policy logic.

Our work lies at the intersection of classical planning and modern LLM-based agents, demonstrating how generalized planning and hierarchical decomposition can be realized dynamically without symbolic domain models by extracting, evaluating, and reusing executable policy components learned through interaction.

Problem Formulation

We consider a structured sequential decision-making setting organized around a *meta-domain* $\mathcal{M}$ that defines a shared action space and deterministic execution semantics.

Within a meta-domain, we distinguish between *domains* and *task instances*. A domain $\mathcal{D}$ is a collection of related task instances that share latent procedural structure but differ in parameters such as entities, values, or initial conditions. A task instance $\tau = (I, s_0, s_g)$ consists of:

- I: a natural-language instruction describing the desired behavior,
- s_0: an initial state,
- s_g: a goal specification defining correctness.

Following generalized planning terminology, we distinguish between policies and plans. A **policy** π is a parameterized program with signature $\sigma(\pi)$ that maps task parameters to executable plans. To solve a task instance τ, the system must: (1) extract parameters θ from τ that match $\sigma(\pi)$, (2) generate a **plan** $\pi(\theta)$ by instantiating π with θ, and (3) submit the plan along with s_0 and s_g to a **validator**. The validator executes $\pi(\theta)$ from s_0 and checks whether the resulting state satisfies s_g. The validator provides feedback indicating success or failure, and may include error messages or execution traces to guide debugging.

The key structural property of this setting is that task instances within a domain share underlying workflows or procedural patterns, differing primarily in parameter values. This structure enables learning parameterized policies that generalize across instances within a domain, and extracting reusable skills that transfer across domains within the same meta-domain.

Objective

Given domains $\{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ within meta-domain $\mathcal{M}$, we synthesize parameterized policies that solve task instances efficiently and compositionally.

For each domain $\mathcal{D}_i$, we seek a parameterized policy π_i with signature $\sigma(\pi_i)$ that solves all task instances in $\mathcal{D}_i$. The policy captures shared procedural structure, while the signature defines varying parameters. Instantiating π_i with task-specific parameters θ_j produces a plan satisfying task requirements.

Beyond solving individual domains, we leverage experience from solved domains to solve new domains efficiently by identifying and reusing recurring procedural patterns. The challenge is learning policies that generalize across task instances within domains and compose reusable knowledge across domains, enabling efficient adaptation without complete policy resynthesis.

AppWorld as a Meta-Domain Instantiation

We instantiate this general formulation on the AppWorld benchmark (Trivedi et al. 2024), which provides a concrete realization of the meta-domain structure.

Meta-domain: The AppWorld benchmark defines the meta-domain $\mathcal{M}$. The environment consists of a set of applications $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ (e.g., Gmail, Spotify, Phone), where each application a_i exposes a collection of APIs denoted by $\mathrm{API}(a_i)$. The global world state s is represented as a database encoding the state of all applications. The agent cannot access or modify s directly; all interaction occurs exclusively through API calls, which deterministically update the world state.

Domains: In AppWorld, each *scenario* corresponds to a domain $\mathcal{D}$. A scenario is a set of tasks $\{\tau_1, \tau_2, \ldots, \tau_k\}$, where typically $k = 3$. Tasks within a scenario invoke the same applications and APIs, and differ primarily in parameter values such as entities, amounts, dates, or labels. For example, a payment scenario may contain multiple tasks that differ only in recipients, amounts, or notes, while sharing the same high-level workflow.

Task instances: Individual tasks within a scenario correspond to task instances. Each task $\tau = (I, s_0, s_g)$ requires the system to generate a parameterized policy (a Python function) and instantiate it with task-specific parameters to produce an executable plan. The plan consists of a sequence of API calls and glue code (such as processing responses, manipulating results, etc). Task correctness is evaluated by executing the plan in the AppWorld simulator and comparing the resulting state to the expected effects encoded in s_g through per-task test cases.

This instantiation makes AppWorld a natural testbed for evaluating generalized and compositional policies: scenarios provide natural domain boundaries, tasks within scenarios share procedural structure, and the benchmark includes multiple scenarios that exhibit recurring patterns across different applications and workflows.

Approach

We present a meta-domain-agnostic architecture for dynamic policy learning that is fully transferable across different problem settings. The approach makes minimal assumptions about the underlying meta-domain: it requires only that policies can be represented as executable programs, that a

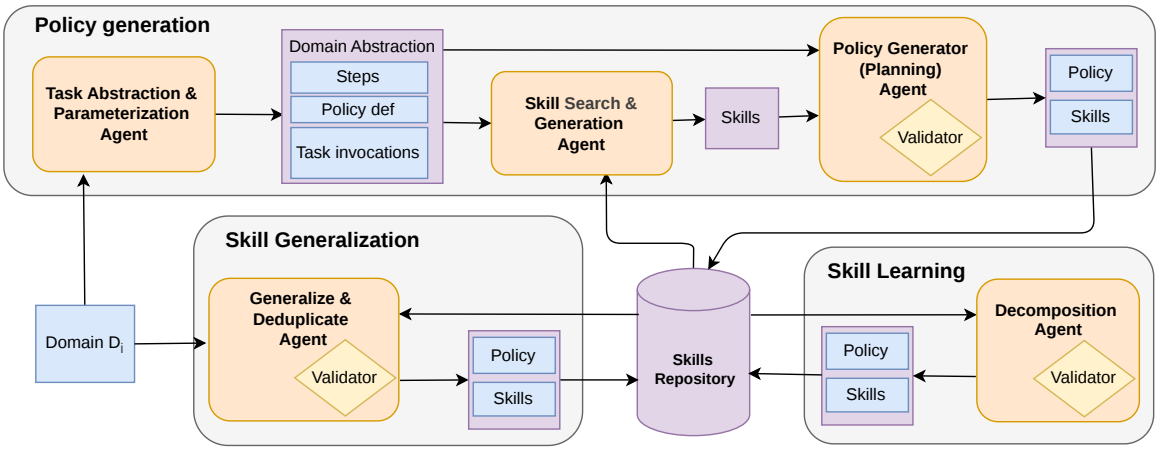

Figure 1: Dynamic policy-learning architecture with three main sections: (1) **Policy generation** synthesizes domain-level policies using task abstraction, skill search, and iterative validation with debugging; (2) **Skill Learning** extracts reusable skills from successful policies via the Decomposition Agent; (3) **Skill Generalization** consolidates and validates generalized skills through the Generalize and Deduplicate Agent, with its own validation loop, before storing them in the validated skill repository for use in subsequent attempts.

validator can check task correctness, and that the validator provides interpretable feedback. The architecture does not depend on specific action spaces, state representations, or symbolic domain models. Instead, it relies on agents capable of semantic understanding and code generation from natural language descriptions, making the system adaptable to different meta-domains without architectural modification.

Figure 1 provides an overview of the architecture and the interactions among its components. Agents are marked by orange boxes in the figure. The architecture consists of three main sections that operate on a given domain: (1) **Policy generation** synthesizes a parameterized policy through task abstraction, skill retrieval, and iterative validation with debugging; (2) **Skill Learning** extracts reusable skills from the successful policy via decomposition and validates them; (3) **Skill Generalization** consolidates learned skills with previously validated ones through clustering and generalization, with its own validation loop to ensure correctness is preserved. The validated skill repository is used for skill retrieval in subsequent domains.

Policy Generation

Policy synthesis proceeds through the flow shown in the policy generation section of Figure 1. Given a domain $\mathcal{D}_i$, the **Task Abstraction and Parameterization Agent** induces a high-level representation capturing shared structure across task instances. This abstraction produces three key artifacts: (1) high-level steps describing the workflow, (2) a policy signature $\sigma(\pi)$ specifying the parameters and interface of the domain-level policy, and (3) concrete parameter bindings $\{\theta_1, \ldots, \theta_k\}$ for each task instance. This abstraction enables generalization across task instances within the domain and guides downstream reasoning. The abstraction process is meta-domain-agnostic: it operates purely on natural language descriptions and does not require knowledge of the specific action space or state representation.

To support compositional policy generation, the **Skill Search Agent** retrieves relevant reusable skills from the validated skill repository. Specifically, all validated skills are embedded and indexed, and only the top-k most relevant ones are retrieved for a given domain based on semantic similarity to the domain abstraction. The retrieved skills, restricted to their signatures and usage information, are provided as context to the **Policy Generator (Planning) Agent**, which is then tasked with generating a parameterized policy π for the domain in the appropriate executable representation (e.g., Python functions for AppWorld). The Policy Generator receives four inputs: the high-level steps, the policy signature $\sigma(\pi)$, the parameter bindings, and the relevant skills from the Skill Search Agent.

The synthesized policy is instantiated with task-specific parameters to produce plans, which are submitted to the validator for evaluation against the task requirements. If validation fails, error messages and execution traces are returned to the Policy Generator Agent, which enters a debugging loop to revise the policy. This validation-debug cycle continues until successful plans are obtained for all task instances or a fixed debugging budget is exhausted. This feedback loop is meta-domain-agnostic: it requires only that the validator provides interpretable error signals.

Skill Learning

When a domain-level policy succeeds, the **Decomposition Agent** analyzes the policy and extracts reusable skills corresponding to coherent fragments of executable logic that may be applicable to other domains. The Decomposition Agent operates through code analysis, identifying procedural patterns and abstractions without requiring domain-

specific heuristics. The agent produces two outputs: (1) learned skills representing the extracted reusable fragments, and (2) an updated policy that references these skills. Both outputs are validated by instantiating the updated policy with the original parameter bindings and submitting the resulting plans to the validator across all task instances within the domain. Upon successful validation, the learned skills are stored in the learned skill repository, while the updated policy replaces the original. At this stage, skills are accumulated but have not yet been generalized or deduplicated.

Skill Generalization

The system evaluates and generalizes skills from the learned repository together with previously validated skills (Figure 1). This stage achieves transfer across domains within the meta-domain. The generalization process can be triggered after processing a fixed number of domains, when the repository reaches a size threshold, or periodically.

Candidate skills are clustered based on functional similarity using semantic embeddings of executable code. Clustering identifies groups of related skills and separates distinct ones, improving efficiency and quality of generalization.

The **Generalize and Deduplicate Agent** consolidates redundant or overly specific skills within each cluster into general abstractions. For example, `login_to_phone` and `login_to_venmo` might merge into `login_to_app`. The agent produces: (1) generalized skills and (2) updated policies using these skills.

Both outputs are validated before acceptance. Updated policies are instantiated with original parameters and submitted to the validator with corresponding task instances, ensuring generalization preserves correctness. If validation fails, the agent enters a debugging loop until policies pass all tests or a budget is exhausted. Successful policy components are stored in the validated repository for retrieval in subsequent domains.

This stage consolidates reusable knowledge and induces hierarchical structure over executable skills. The validated repository provides a growing library supporting efficient compositional policy generation.

All agents operate through semantic understanding and code generation, making the approach representation-agnostic. The architecture adapts to different meta-domains by learning appropriate executable representations through validator interaction, without requiring manual domain models. This enables application to web automation, database management, API orchestration, or other structured settings with executable policies and interpretable feedback.

Experimental Evaluation

We aim to answer three key research questions:

1. **Efficiency of reuse:** Does dynamic learning and reuse of policy components reduce the computational effort required to solve new task instances compared to synthesizing policies from scratch?
2. **Cost-accuracy trade-offs:** How does the proposed approach balance inference cost against task success, particularly in resource-constrained settings?
3. **Generalization across domains:** Can skills learned from one set of scenarios transfer effectively to previously unseen scenarios, including those involving different applications?

To address these questions, we evaluate our approach on the AppWorld benchmark, which provides a structured environment for multi-step decision-making with shared procedural patterns across scenarios. We measure performance not only in terms of final task success, but also as a function of debugging iterations and cumulative inference cost, providing insight into the efficiency gains from dynamic reuse.

AppWorld Implementation Details

Policy Generation. Policy synthesis is guided by semantic retrieval over both APIs and reusable skills. All validated skills are embedded and indexed using the `BAAI/bge-large-en-v1.5` (BAAI) embedding model. For each domain (scenario in AppWorld), the Skill Search Agent retrieves the top-k most relevant reusable skills based on cosine similarity to the domain abstraction. We set $k = 20$ for skill retrieval. The Policy Generator Agent also retrieves the top-$k = 20$ most relevant APIs from the AppWorld API documentation using the same embedding model and similarity metric.

Skill Learning. The Decomposition Agent analyzes successful domain-level policies to extract reusable skills. In the AppWorld instantiation, the agent identifies Python functions that implement coherent subtasks (e.g., authentication workflows, data retrieval patterns, or transaction sequences). The agent produces both the extracted skills and an updated policy that references them. Validation ensures that the updated policy produces identical behavior to the original across all task instances in the domain.

Skill Generalization. Skill clustering uses the BAAI embedding model to compute pairwise cosine similarity over all candidates from the learned repository. We apply threshold-based clustering with $\tau = 0.85$, followed by a greedy clustering procedure to obtain initial clusters. The Generalize and Deduplicate Agent processes each cluster to consolidate redundant or overly specific skills. For example, in AppWorld, the agent may merge domain-specific authentication functions like `login_to_phone` and `login_to_venmo` into a single parameterized `login_to_app`. The agent can also subsume entire clusters by producing higher-level abstractions. Both the generalized skills and updated policies are validated against all original task instances before being accepted into the validated skill repository.

Execution and Validation. Each generated policy is executed within the AppWorld environment, which provides a sandboxed Python runtime with access to simulated applications and APIs. The execution environment captures all API calls, return values, and any runtime errors or exceptions. Validation is performed by comparing the policy's execution trace and final state against the ground-truth goal conditions specified for each task instance. The environment then reports success or failure.

For domain-level policies, we require success across all task instances within the scenario. When learning or generalizing skills, the system validates success of the updated policies (which reference the new skills). We set the debugging budget to three iterations per validation cycle, allowing the Policy Generator Agent to iteratively refine policies based on execution feedback (error messages, API responses, or partial results) before a policy is deemed unsuccessful.

Prompts and Implementation. The prompts used for LLM calls to the Task Abstraction and Parameterization Agent, Policy Generator Agent, Decomposition Agent, and Generalize and Deduplicate Agent are provided in the supplementary material.

Experimental Setup

Benchmark. We use the AppWorld benchmark (Trivedi et al. 2024), which consists of scenarios organized into training, normal test, and challenge test sets. Each scenario contains multiple related tasks that share underlying workflows but differ in parameters. The challenge test set includes applications (Gmail, Amazon) that do not appear in the training set, enabling evaluation of cross-application transfer.

AppWorld defines two primary evaluation metrics:

- **Task Goal Completion (TGC):** Success is measured independently per task, providing a granular view of performance on individual task instances.
- **Scenario Goal Completion (SGC):** A scenario is counted as successful only if all tasks within the scenario are solved. This metric directly evaluates the ability to synthesize policies that generalize consistently across task instances within a scenario.

We report both metrics where applicable, and analyze performance as a function of debugging iterations and cumulative inference cost to assess efficiency alongside accuracy.

Our Method. We evaluate our dynamic policy-learning architecture as described in the previous section, which includes all components: Task Abstraction and Parameterization Agent, Skill Search Agent, Policy Generator Agent, Decomposition Agent, and Generalize and Deduplicate Agent. The architecture is model-agnostic and operates through semantic understanding and code generation from natural language descriptions.

To obtain the initial seed for the validated skill repository, we run the full architecture on the 30 scenarios in the AppWorld training set, starting from an empty skill repository. For each training scenario, we perform policy generation and skill learning as described in the previous section. After processing all training scenarios, the learned skill repository contains skills extracted from successful policies across the training set. We then invoke the generalizing skills stage, which clusters, deduplicates, and generalizes the learned skills. This process produces a validated skill repository containing 159 reusable skills. For these steps, we use the Claude model. While Claude successfully solves all 30 scenarios, the first iteration produces 101 reusable skills, and the second iteration increases this number to 159. We use the resulting skill repository to seed the experiments on both the normal and challenge test sets, regardless of the model used on the test set. This design choice enables controlled comparison of how different models leverage the same initial knowledge base during test-time learning, isolating the contribution of online skill learning and reuse independent of training-time differences. This seeded repository is then used as the starting point for all experiments on the test sets (normal and challenge), enabling the system to leverage previously learned knowledge when encountering new scenarios. During test set evaluation, new skills continue to be learned from successful policies and accumulated in the learned skill repository. After processing a fixed number of debugging iterations (20 in our experiments), the generalization process is invoked again to consolidate newly learned skills with the existing validated repository, further extending the set of reusable skills available for subsequent scenarios.

Method	Model	Normal		Challenge	
		TGC	SGC	TGC	SGC
Claude Code	Claude Opus 4.5	66.0	56.6	–	–
Smolagents Code	Claude Opus 4.5	70.0	60.4	–	–
OpenAI Solo	Claude Opus 4.5	68.0	56.6	–	–
ReAct	Claude Opus 4.5	61.0	52.8	–	–
ReAct Short	Claude Opus 4.5	64.0	54.7	–	–
Alibaba AgentRL	Qwen3-14B	86.9	80.4	67.6	50.4
IBM CUGA	GPT-4.1	73.2	62.5	57.6	48.2
LOOP	Qwen2.5-32B	72.6	53.6	47.2	28.8
GP (our baseline)	GPT-OSS 120B	0.0	0.0	0.7	0.7
GP (our baseline)	Claude Sonnet 4.6	96.4	96.4	83.2	82.0
HCL-GP	GPT-OSS 120B	62.5	62.5	41.0	41.0
HCL-GP	Claude Sonnet 4.6	**98.2**	**98.2**	**98.3**	**97.8**

Table 1: Comparison of our scenario-level policy synthesis approach with task-level baselines on the AppWorld normal and challenge test sets. The challenge set includes applications (Gmail, Amazon) not present in the training set.

Model Configurations. We evaluate with two language models: **Claude Sonnet 4.6** (frontier model with strong code generation) and **GPT OSS 120B** (open-source model at different capability level). We compare our HCL-GP configuration (with skill retrieval, learning, and reuse) against a GP configuration described below.

GP Baseline. The GP baseline implements generalized planning without skill learning or reuse. It includes the Task Abstraction and Parameterization Agent and the Policy Generator Agent with the same validation and debugging loop as HCL-GP. However, it does not include the Skill Search Agent, Decomposition Agent, or Generalize and Deduplicate Agent. The baseline starts each scenario with an empty context and does not accumulate or reuse skills across scenarios. This isolates the contribution of hierarchical skill learning and reuse from the core generalized planning approach.

Additional baselines. We include results from the AppWorld leaderboard (top three task-level methods: Alibaba

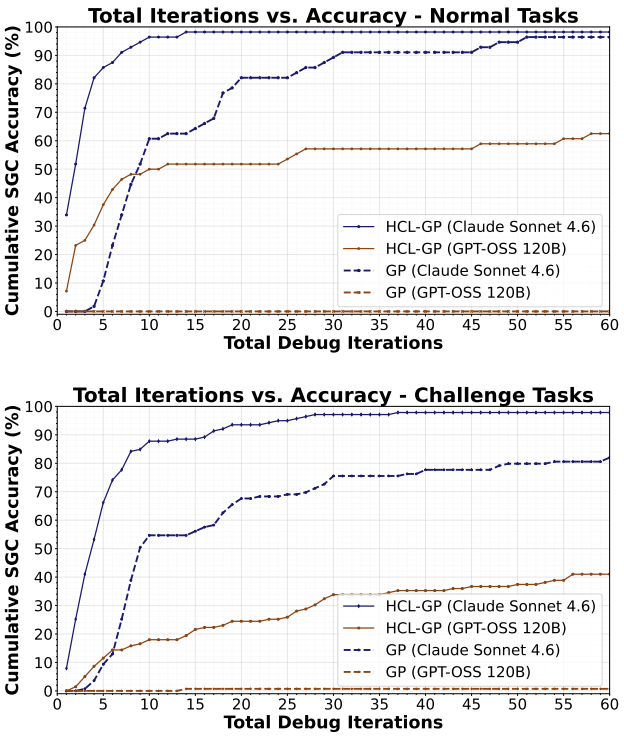

Figure 2: Cumulative SGC per total debugging iterations on the AppWorld normal (top) and challenge (bottom) test sets.

Cloud ApsaraLab AgentRL using Qwen3-14B, IBM CUGA (Marreed et al. 2025) using GPT-4.1, and LOOP (Chen et al. 2025) using Qwen2.5-32B) and agent architectures that use Claude Opus 4.5 from Bandel et al. (2026).

Table 1 summarizes results. The leaderboard methods and Bandel et al. methods operate at the task level rather than synthesizing scenario-level policies, resulting in lower SGC than TGC. In contrast, our dynamic policy-learning architecture, as well as our baseline, focuses on generalization across scenarios and improving efficiency through reusable skills. The slight increase in TGC occurs when the policy computed in the final debugging iteration solves some but not all of the tasks within a scenario.

Any-time Results

Rather than reporting only final task success, our evaluation emphasizes how quickly success is achieved as a function of debugging effort and inference cost. This perspective is particularly appropriate for dynamic and interactive systems, where efficiency and reuse play a central role.

Iteration Efficiency. Figure 2 shows cumulative SGC as a function of total debugging iterations on the normal and challenge test sets. We compare our proposed approach HCL-GP against the baseline variant GP for both Claude and GPT-OSS models. As a reminder, after 20 and then 40 iterations HCL-GP performs generalization of freshly learned and previously existing skills, and updates the set of validated skills.

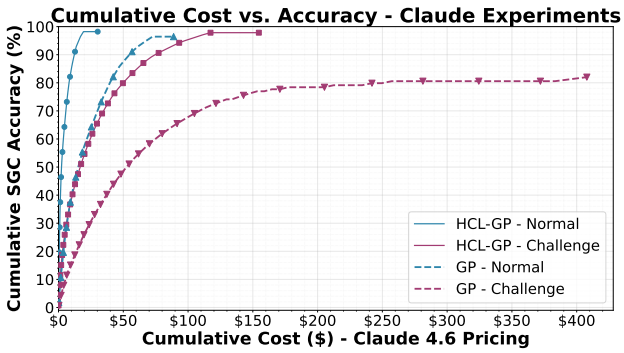

Figure 3: Cumulative SGC as a function of cumulative inference cost for using Claude 4.6 pricing.

The results demonstrate that all methods benefit from additional debugging iterations, confirming that execution feedback and iterative repair are effective in this setting. However, the efficiency with which methods convert debugging effort into solved scenarios varies substantially. While Claude consistently outperforms GPT-OSS across all configurations, the largest performance gap appears between the GP and HCL-GP variants for GPT-OSS, where GP achieves near-zero success (0.0% on Normal, 0.7% on Challenge), demonstrating that without skill reuse, the open-source model cannot effectively solve any of these scenarios.

For Claude, the GP variant achieves strong performance (96.4% on Normal, 82.0% on Challenge), though HCL-GP reaches even higher success rates (98.2% and 97.8% respectively) while requiring fewer debugging iterations. In contrast, HCL-GP consistently attains higher success earlier in the iteration budget, demonstrating that reusable skills accelerate convergence. This advantage is particularly striking on the challenge set, where HCL-GP improves upon GP by 15.8 percentage points (82.0% $\rightarrow$ 97.8%), compared to only 1.8 points on the normal set. This substantial gap demonstrates that learned skills provide critical support when encountering unfamiliar applications.

The HCL-GP approach dramatically improves GPT-OSS performance from near-zero to 62.5% on Normal and 41.0% on Challenge tasks. However, this still falls well short of Claude's performance, suggesting that while skill reuse is essential for open-source models, base model capability remains a significant factor.

Overall, HCL-GP policies maintain a consistent advantage over their GP counterparts, solving more scenarios with fewer iterations. Notably, the challenge dataset includes applications such as Gmail and Amazon that do not appear in the training set. As a result, the initial skill repository does not contain application-specific skills for these applications. Despite this, HCL-GP benefits from previously learned generalized skills that transfer across applications, contributing to improved performance on unseen domains.

Cost-Accuracy Trade-offs. For cost analysis, we report cumulative inference cost using the pricing of Claude Sonnet 4.6 ($3 per million input tokens and $15 per million out-

put tokens). Figure 3 reports cumulative SGC as a function of cumulative inference cost for the Claude-based configurations.

Across all settings, success increases rapidly at low cost and exhibits diminishing returns as cost grows. Dynamic reuse shifts this trade-off favorably: higher success is achieved earlier in the cost budget, particularly on challenge tasks. This result reinforces the interpretation from the iteration-based analysis. Dynamic reuse does not merely improve final success, but more efficiently converts inference budget into solved tasks by reducing redundant policy synthesis and debugging.

The cost analysis is particularly relevant when comparing against task-level methods. While we do not have detailed cost breakdowns for all baseline methods, the fundamental difference in approach suggests that scenario-level policy synthesis with reuse should be more cost-effective: rather than solving each task independently (potentially repeating similar reasoning across related tasks), our approach synthesizes a single parameterized policy per scenario and reuses learned skills across scenarios.

Summary of Key Findings. Taken together, these results highlight three key observations:

1. **Scenario-level synthesis is effective:** Even without dynamic reuse, synthesizing parameterized policies at the scenario level can substantially outperform task-level methods in terms of SGC, as demonstrated by Claude's 96.4% baseline. However, this advantage depends on sufficient base model capability, as GPT-OSS's baseline performance (0.0%) falls below task-level methods.
2. **Dynamic reuse improves efficiency:** The advantages of reusable skills are most pronounced in terms of iteration efficiency and cost-accuracy trade-offs. Dynamic reuse enables the system to achieve high success rates earlier in the debugging budget, particularly on challenging scenarios.
3. **Skills transfer across applications:** Learned skills generalize beyond the specific applications they were extracted from, as evidenced by improved performance on challenge scenarios that include previously unseen applications (Gmail, Amazon).

These findings support the view that reusable, compositional structure, learned dynamically from prior executions, plays a critical role in scaling interactive agents to complex, multi-step domains. The combination of scenario-level policy synthesis and dynamic skill learning provides both higher final accuracy and more favorable efficiency characteristics compared to task-level approaches.

Search Effectiveness

In this ablation study, we evaluate the *Search* component on the 30 training scenarios under different embedding models and retrieval configurations. We evaluate both FAISS-based approximate nearest-neighbor search (Johnson, Douze, and Jégou 2021) and an OpenSearch-based retrieval backend https://opensearch.org/ for semantic search.

Configuration	MRR	MAP	R@5	R@10	R@20
OpenSearch + BAAI	0.24	0.43	35.2	51.4	70.2
FAISS + BAAI	0.24	0.44	36.2	56.5	68.3
FAISS + Nomic	0.21	0.35	29.0	47.7	63.8

Table 2: API retrieval performance on the AppWorld training set for different embedding and retrieval configurations. Metrics include mean reciprocal rank (MRR), mean average precision (MAP), and recall at k (R@5, R@10, R@20).

Metric	**Normal (S/M/L)**	**Challenge (S/M/L)**
Total Helpers Used	42 / 1 / 203	46 / 2 / 638
Utilization Rate %	26 / 20 / 75	29 / 40 / 60
Helpers per Scenario	2.9 / 0.8 / 3.8	1.5 / 0.9 / 6.4
Reuse Rate	3.7 / 46.0 / 1.0	4.5 / 64.5 / 1.4
Multi-use %	59.5 / 100 / 1.5	58.7 / 100 / 11.1

Table 3: Skill usage statistics. **Total**: unique helpers used. **Utilization Rate**: % of available used. **Helpers per Scenario**: mean per scenario. **Reuse Rate**: mean scenarios per helper. **Multi-use**: % used in 2+ scenarios.

Note that Nomic is short for the *nomic-ai/nomic-embed-text-v1.5* embedding model. For the training set, AppWorld provides ground-truth annotations specifying the APIs required to solve each task. Using this information, we evaluate the Search component in isolation by measuring its ability to retrieve relevant APIs.

Table 2 reports standard information retrieval metrics: mean reciprocal rank (MRR), mean average precision (MAP), recall at 5 (R@5), recall at 10 (R@10), and recall at 20 (R@20). Recall values indicate the fraction of tasks for which at least one correct API appears within the top-k retrieved results. Based on these results, we select a FAISS-based configuration using the BAAI embedding model for both semantic search and reusable skill clustering in all experiments.

Table 2 shows that FAISS-based retrieval with BAAI achieves the strongest recall at lower cutoffs (R@10 and R@5), while maintaining competitive MRR and MAP. Although OpenSearch slightly outperforms FAISS in R@20, FAISS provides more balanced recall across cutoffs, with better R@5 and R@10.

Skill Usage Analysis

Table 3 summarizes skill usage patterns across test sets. Skills are categorized as Seed unchanged (S), Seed modified (M), and Learned (L).

Three key patterns emerge. First, learned skills show high utilization (60–75%) compared to seed skills (26–29%), indicating the system learns task-relevant skills. Second, the two modified skills are used directly or indirectly (through other skills) across scenarios, with `get_supervisor_credentials` used in all scenarios. Seed skills provide stable reuse, while learned skills are mostly task-specific. Third, Challenge scenarios require more learned skills per scenario (6.4 vs 3.8), and the 3×

growth in unique learned skills (203→638) exceeds the 2.5× dataset size increase (56→139 scenarios), indicating more diverse skill learning on harder tasks.

Challenges and Discussions

The results highlight several insights about dynamic reuse and policy generalization in structured interactive environments. First, the benefits of reusable skills are strongly task-dependent. On simpler scenarios, execution feedback and iterative debugging alone is often sufficient to recover from initial policy errors, leading to convergence even without reuse. In these settings, the primary contribution of reusable skills is improved efficiency, enabling successful policies to be synthesized with fewer debugging iterations. In contrast, on more challenging scenarios, reusable skills play a more substantive role, providing structural guidance that helps the agent avoid repeated errors and recover from failures more effectively. This suggests that dynamic reuse is particularly valuable in domains characterized by longer execution horizons or more brittle workflows.

A second observation concerns the nature of generalization. Reusable skills learned from one set of scenarios can transfer to unseen applications, as demonstrated by improved performance on challenge scenarios that include applications not present in the training set. This indicates that the learned skills capture procedural abstractions that extend beyond app-specific logic. For example, authentication patterns learned from Phone and Venmo generalize to Gmail and Amazon, suggesting that the system identifies workflow-level abstractions rather than application-specific implementations. At the same time, this form of generalization is opportunistic rather than guaranteed: it emerges through execution grounding and consolidation rather than through explicit domain modeling. Understanding the conditions under which such transfer is reliable remains an open research question.

The architecture also exposes several challenges that become increasingly important when moving beyond AppWorld. One challenge concerns assumptions about the availability of a well-defined *meta-domain*. In AppWorld, the action space, execution semantics, and validation mechanism are fixed and explicitly provided. In more general settings, such a meta-domain may be only partially specified or may evolve over time, requiring the system to adapt to changing interfaces or infer validation criteria from less structured feedback. Supporting such adaptation would be necessary to deploy similar architectures in less structured environments.

A related challenge arises from the clean scenario structure provided by AppWorld. Scenarios group together tasks that share underlying structure, enabling straightforward learning of scenario-level policies. In many real-world settings, however, such scenario boundaries may not be given a priori. Identifying which tasks belong to the same domain or which tasks are sufficiently similar to benefit from shared abstractions becomes a nontrivial problem. Addressing this challenge would require mechanisms for clustering or linking tasks based on inferred procedural structure, execution traces, or learned representations, rather than relying on externally provided scenario labels.

Scalability of reusable skill evaluation presents another challenge. Although clustering provides a coarse mechanism for grouping similar skills, the subsequent generalization and deduplication stages rely on LLM-based reasoning that is sensitive to prompt length and context limits. As the number of skills grows, managing the token footprint of these operations becomes increasingly important. More structured multi-stage evaluation pipelines, or incremental abstraction mechanisms, may be needed to sustain long-term learning.

Taken together, these challenges delineate the boundary between the capabilities demonstrated in AppWorld and the requirements for deploying dynamic reusable policy learning in broader decision-making settings. Addressing them will be essential for extending this line of work beyond well-structured benchmarks to more open-ended and less curated environments.

Summary and Future Work

We presented a dynamic policy-learning architecture for structured interactive environments that integrates ideas from generalized planning and hier. decomposition with LLM-based agents. The approach operates at two levels: a per-scenario flow that synthesizes parameterized scenario-level policies, and an attempt-level process that extracts, evaluates, and generalizes reusable executable skills. By separating execution, learning, and generalization, the system incrementally constructs a reusable skill repository that supports compositional policy generation across scenarios.

Empirical results on the AppWorld benchmark demonstrate that dynamic reuse improves both iteration efficiency and cost–accuracy trade-offs, particularly on challenging scenarios that require longer and more structured decision-making. The results also highlight that reusable skills need not be tied to specific applications: generalized skills learned from one set of scenarios can transfer to previously unseen ones, supporting broader generalization within a shared meta-domain.

There are several promising directions for future work. First, while reusable skills are currently treated as flat executable units, richer structural organization could be explored, such as learning explicit dependency graphs or multi-level abstractions over policy components. Second, the current approach relies primarily on semantic similarity for clustering, with generalization and deduplication combined into a single LLM invocation. While clustering itself could be further improved, separating the generalization and deduplication stages would substantially reduce the token footprint of each LLM call. In practice, we have observed that the combined operation can exceed the maximum context length of the underlying model, making this separation an important direction for improving scalability. Finally, applying the architecture to domains beyond AppWorld would help assess the broader applicability and impact of the proposed approach.

Overall, this work suggests that dynamically learned, reusable executable skills provide a practical bridge between classical planning principles and modern LLM-based agents, enabling efficient and compositional problem solving in realistic interactive settings.

References

Argall, B. D.; Chernova, S.; Veloso, M. M.; and Browning, B. 2009. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5): 469–483.

Bandel, E.; Yehudai, A.; Eden, L.; Sagron, Y.; Perlitz, Y.; Venezian, E.; Razinkov, N.; Ergas, N.; Ifergan, S. S.; Shlomov, S.; Jacovi, M.; Choshen, L.; Ein-Dor, L.; Katz, Y.; and Shmueli-Scheuer, M. 2026. General Agent Evaluation. arXiv:2602.22953 [cs.AI].

Bonet, B.; Giacomo, G. D.; Geffner, H.; and Rubin, S. 2017. Generalized Planning: Non-Deterministic Abstractions and Trajectory Constraints. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 873–879. IJCAI.

Chen, K.; Cusumano-Towner, M.; Huval, B.; Petrenko, A.; Hamburger, J.; Koltun, V.; and Krähenbühl, P. 2025. Reinforcement Learning for Long-Horizon Interactive LLM Agents. arXiv:2502.01600 [cs.LG].

Erol, K.; Hendler, J. A.; and Nau, D. S. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994)*, 1123–1128. AAAI Press.

Georgievski, I.; and Aiello, M. 2015. HTN Planning. *Artificial Intelligence*, 222: 124–156.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Hodel, N. 2024. *Exploring the Use of LLMs in Generalized Planning*. Bachelor's thesis, Saarland University.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 950–956. AAAI Press.

Hutsebaut-Buysse, M.; Mets, K.; and Latré, S. 2022. Hierarchical Reinforcement Learning: A Survey and Open Research Challenges. *Machine Learning and Knowledge Extraction*, 4(1): 172–221.

Jiménez, S.; Segovia-Aguas, J.; and Jonsson, A. 2019. A Review of Generalized Planning. *The Knowledge Engineering Review*, 34: e5.

Johnson, J.; Douze, M.; and Jégou, H. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 7(3): 535–547.

Marreed, S.; Oved, A.; Yaeli, A.; Shlomov, S.; Levy, I.; Sela, A.; Adi, A.; and Mashkif, N. 2025. Towards Enterprise-Ready Computer Using Generalist Agent. arXiv:2503.01861 [cs.DC].

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20: 379–404.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J.; Pack Kaelbling, L.; and Katz, M. 2024. Generalized Planning in PDDL Domains with Pretrained Large Language Models. In Dy, J.; and Natarajan, S., eds., *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2024)*, 20256–20264. AAAI Press.

Stein, K.; Hodel, N.; Fišer, D.; Hoffmann, J.; Katz, M.; and Koller, A. 2026. Improved Generalized Planning with LLMs through Strategy Refinement and Reflection. In *Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)*. AAAI Press.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112: 181–211.

Trivedi, H.; Khot, T.; Hartmann, M.; Manku, R.; Dong, V.; Li, E.; Gupta, S.; Sabharwal, A.; and Balasubramanian, N. 2024. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. In Ku, L.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 16022–16076. Association for Computational Linguistics.

Appendix

Agent Prompt Specifications

This appendix documents the prompts used for the Policy Generator Agent, Listing 1 and 2, Task Abstraction and Parameterization Agent, Listing 3, Decomposition Agent, Listing 4, and Generalize and Deduplicate Agent, Listing 5.

Listing 1: Policy Generation Agent prompt used to synthesize scenario-level policies.

```
# AI Coding Agent for AppWorld Policy Generation

You are an AI coding agent specializing in generating generalized policy in Python code. Your goal is to design a
**generalized policy (in Python code)** for each scenario (collection of similar tasks) that can solve any of the
tasks under that scenario. The tasks are all from the Appworld benchmark (https://appworld.dev/task-explorer).

Each policy must:
* Handle all possible points of failure (e.g., insufficient balance, expired cards, authentication issues).
* Be modular | reusable components separated from scenario-specific logic.
* Be self-contained, robust, and executable within AppWorld.
* Assume that all tasks are solvable, though they may include realistic hurdles or distractors.

## Resources to Use
### 1. All APIs in AppWorld
To get a list of apps:
```python
print(apis.api_docs.show_app_descriptions())
```

Output:
```
["amazon : An online shopping platform where users can buy a wide variety of products.",
 "spotify : A music streaming platform that provides access to a vast library of songs, albums, and playlists.",...]
```
2. Available Reusable Components
The reusable components are organized into the following categories. **The full implementation code for these
functions is provided separately - do not rewrite them, use them exactly as provided.**

Category Overview:
1. **API Discovery** - Find and understand unknown APIs
2. **Learned API Utilities** - Pre-discovered components you can call directly - You will be provided with the most
relevant ones for your task to start with. You can implement new components based on the task at hand.
3. **Recommended APIs** - Recommnded APIs relevant to your task.

CATEGORY 1: API Discovery
You can discover APIs using:
```python
# Search for APIs
search_results = apis.api_docs.search_api_docs(
    query="your search terms",
    page_limit=20)

# Show descriptions for all APIs available for a given app
api_descriptions = apis.api_docs.show_api_descriptions(
    app_name="app_name")

# Get API structure
api_info = apis.api_docs.show_api_doc(
    app_name="app_name",
    api_name="api_name")
```
Important Notes on API Discovery:
- Do NOT guess API names like `list_contacts`, `get_contacts`, `find_contacts`, `get_supervisor_info`, `list_accounts`,
`get_user_account_info` without checking first - these don't exist
- Always use `search_api_docs()` or `show_api_descriptions()` to discover available APIs
- Check the API structure with `show_api_doc()` before calling it as it may require additional required fields, such
as access_token
- Different apps may have different parameter names (e.g., `relationship` vs `contact_type`)
- For apps that require pagination (e.g., `page_limit`), handle them in your code so you consider all results.
```

Listing 2: Policy Generation Agent prompt (continued)

```
## CATEGORY 2: Learned API Utilities (Includes Dynamically Loaded Suggestions for This Scenarion - Try to use them to
your maximal extent, and implement new components that you deem fit for the task at hand.)
The following pre-filtered utilities are ranked from highest to lowest relevance for your scenario.

{{SHORTLISTED_UTILITIES}}

## CATEGORY 3: Recommended APIs
The following are ranked APIs from highest to lowest relevance to your task.

**Important:** These are the most relevant APIs for your specific task. Start by exploring these APIs first before
searching for alternatives.
**When to use**: For operations not covered by learned utilities but semantically relevant to your task. The
top-ranked APIs are most relevant to accomplishing the task.

{{SUGGESTED_APPWORLD_APIS}}

**How to use**:
Call directly if you're confident
   - `apis.spotify.remove_song_from_library(song_id=X, access_token=Y)`
   - But you must handle errors yourself

## Requirements
### 1. Solvability. All tasks are designed to be solvable. Consider all contingencies for API call outputs.
### 2. Number Answers. Convert answers to nearest integer. Answer format: `answer=10` not "ten".
### 3. Code Style
* Maintain syntactically valid, readable Python code
* Use `apis.<app>.<endpoint>()` if calling an end point directly
* Integrate API calls with control flow
### 4. Error Handling. While reusable components handle many errors internally, your scenario-specific code should:
* Check for empty list returns from reusable functions
* Check for `None` returns when fetching data
* Handle exceptions when calling APIs that aren't wrapped in reusable components
* Implement appropriate logic (retry, skip, or terminate) based on requirements
### 5. Progress Tracking & Debugging. Include `print()` statements throughout the code:
* Use clear step dividers: `print("="*80)`
* Use status markers: `[OK]`, `[FAIL]`, `[WARNING]`
* Print variable states after important operations
### 6. File System Reference. Use file_system app APIs, never `os` or `shutil`.
### 7. Current Date and Time. Use `datetime.now()` and `get_date_range()`. Never rely on your existing knowledge
of what the current date is.
### 8. Data Selection
* References to people mean phone contacts
* Filter results carefully before selecting
* Do not make assumptions - always verify
### 9. Task-Specific Terminology. Follow exactly the terms, labels, and tags specified in the task.
### 10. Contact Matching Safety. Prefer email-first matching, then exact full-name match only if unique.
### 11. Task Completion
* Call `apis.supervisor.complete_task()` when done
* If task asks for information, pass it as: `apis.supervisor.complete_task(answer=<answer>)`
* Answer should be a valid number, integer, or string only
* You can pass `status="fail"` if you cannot solve the task
### 12. Autonomous Decision Making. Make all decisions autonomously. No clarifications.
### 13. Learned Utilities vs Discovery Pattern

...
**Structure your code like this:**

```python
def scenario_X_policy(param1, param2, ...):
    \"\"\"
    Generalized policy for scenario X.
...
```
```

Listing 3: Task Abstraction and Parameterization Agent prompt used to infer scenario-level parameters and high-level task structure from natural language descriptions.

```
You are defining the callable Python function signature for a scenario policy. Input is a SCENARIO DESCRIPTION
containing multiple tasks from the same scenario. Your job:

1) Propose a minimal, stable list of policy function parameters that the runner can pass at call time.
2) Every policy MUST accept `task_text: str` (required).
3) Do NOT include parameters that cannot be derived from task_text or discovered via AppWorld APIs at runtime.
4) Keep params general across tasks in the scenario (avoid overfitting to one task).
5) IMPORTANT: Based on each of the 3 tasks mentioned in the scenario description, assign param_values to each
parameter you define, except the task_text parameter. This is mandatory.
6) IMPORTANT: Identify which AppWorld apps are needed for this scenario.
Available Apps[
 "amazon : An online shopping platform where users can buy a wide variety of products.",
 "spotify : A music streaming platform that provides access to a vast library of songs, albums, and playlists.",...]
Add an "apps_used" field to your JSON output listing the apps mentioned or implied in the tasks.
7) IMPORTANT: Plan out the high-level steps needed to accomplish the task. What are the steps you will need take
to complete the task?
   - For each step, be as specific and detailed as possible. Include all the steps that are necessary including
   logging in to an app or getting the necessary credentials.
   - ALWAYS explicitly mention which app(s) will be used in that step (e.g., "Use file_system to read...",
   "Use venmo to send...")
   - If a step involves multiple apps, try to break this step into multiple steps, rather than having a step
   that mentions more than one app
   - Use the exact app names from the list above (e.g., "file_system" not "filesystem")
   Add a "steps_needed" field to your JSON output listing all the steps needed to accomplish the task.

Output MUST be valid JSON with schema:

### Example (for guidance only)
Scenario question:
"What is the title of the most-liked song in my Spotify playlists."
A good parameter interface for this scenario would be:
{{"apps_used": ["spotify", "supervisor"],
  "steps_needed": [
    "Get the credentials needed from Supervisor for login",
    "Login to Spotify",
    "Query Spotify for the user's playlists",
    "Find the most-liked song in the remaining playlists",
    "Return the title of the most-liked song"],
  "params": [
    {{"name": "task_text",
      "type": "str",
      "description": "Full natural-language instruction for the current task instance",
      "required": true,
      "default": null,
      "source": "scenario_text",
      "extraction_hint": "Use as-is",
      "param_values": {{}}}},
    {{"name": "include_private_playlists",
      "type": "bool",
      "description": "Whether to include the user's private playlists when interpreting 'my Spotify playlists'",
      "required": false,
      "default": true,
      "source": "scenario_text",
      "extraction_hint": "If scenario_text explicitly excludes private playlists, set false; otherwise default true",
      "param_values": {{"task_1": True, "task_2": False, "task_3": True}}}}]}}

Notes:
- These parameters express TASK INTENT, not API details.
- Optional parameters must always have safe defaults.
- The policy must still work if optional parameters are null.

Allowed types: str, int, float, bool, list[str]
Allowed sources: task_texts, constant, derived
Return JSON only (no extra text).

SCENARIO DESCRIPTION: {scenario_description}
```

Listing 4: Decomposition Agent prompt used to identify and factor reusable components from successfully executed policies.

````
#You are a code extraction agent. Your task is to extract reusable components from a successful policy implementation.

The policy has successfully solved a task. Now extract any reusable components:

**What to Extract:**
- API calls that were discovered and successfully used (wrap them in utility functions)
- Data transformations, validations, calculations
- Filtering patterns, field extraction logic
- Any repeated code blocks or inline logic that could be reused

**Make components general-purpose, not scenario-specific.**

## CRITICAL: Output Format

You MUST provide EXACTLY two sections in this format:

### REUSABLE COMPONENTS
```python
# Component 1 - <Description> (Discovered from Scenario <ID>)

def component_name(params):
    \"\"\"
    Clear docstring explaining what this does.

    Args:
        param1: Description
        param2: Description

    Returns:
        Description of return value
    \"\"\"
    # Implementation

# Component 2 - <Description> (Discovered from Scenario <ID>)

def another_component(params):
    \"\"\"
    Clear docstring.
    \"\"\"
    # Implementation
```

### UPDATED POLICY
```python
def scenario_<id>_policy(params):
    \"\"\"
    Updated policy using the extracted reusable components.
    \"\"\"
    # Implementation using the new components
```

## Rules:
- Use EXACTLY these two headings
- Each section must have ONE ```python code block
- NO text outside these two sections
- NO explanations or commentary
````

Listing 5: Generalize and Deduplicate Agent prompt used to deduplicate, generalize, and update learned component repository across scenarios.

```
#You are a code generalization expert specializing in creating reusable, generic components from specific
implementations. Your task has TWO parts:

## Part 1: Generalize Multi-Function Clusters
Analyze clusters of similar functions learned from different scenarios and create ONE generalized function
per cluster that:
1. Works for all functions in the cluster
2. Has a generic name (e.g., 'authenticate' instead of 'authenticate_venmo')
3. Uses parameters to handle app-specific or scenario-specific differences
4. Includes comprehensive docstring explaining the generalization
5. Maintains all functionality from the original functions

**IMPORTANT - Clustering Flexibility:**
- The initial clustering may not be perfect. You have the flexibility to:
  - **Split a cluster** if functions are too different and shouldn't be merged (keep them as separate functions)
  - **Merge across clusters** if you notice functions from different clusters that should actually be combined
  - **Include helper functions** from other clusters if they would improve the generalized implementation
- Use your judgment: if functions in a cluster are fundamentally different despite similarity scores, keep them
separate and document why in the docstring
- Conversely, if you see opportunities to combine functions across clusters for better reusability, do so

## Part 2: Update Dependent Single Functions (if provided)
Some single functions may call the original functions from Part 1. After generalizing those functions, you MUST:
1. Update these single functions to call your NEW generalized functions
2. Adjust parameters to match the new generalized function signatures
3. Preserve all other functionality in these single functions

## Key Principles:
**Generalization:**
- Remove app-specific names when possible (venmo_login → login, with app parameter). However, if the function is
actually only meant for a particular app, then keep the app name in the name of the function and include the app name
in the docstring.
- Use parameters for variations instead of separate functions
- If the function have different paramaters and different return values but perform similar function, still merge them.
- **IMPORTANT**: You can call other generalized functions you create in your implementations - build on top of
simpler functions
- **IMPORTANT**: DO NOT call the original functions (e.g., original_function_1, original_function_2) - they will
not be available. Only use the generalized functions you create and the AppWorld APIs (apis.*)
**Smart Parameter Design:**
- You do NOT need to accommodate every possible parameter variation from all versions
- Choose the MOST SENSIBLE parameter set that covers the core functionality
- Add an app parameter (e.g., app_name="spotify") to make functions work across different apps
- Don't create overly complex signatures trying to handle every edge case
- Focus on the common use case and let the app parameter handle app-specific differences
**Updating Dependent Functions:**
- When updating single functions that call original functions, replace those calls with your new generalized functions
- Adjust parameters carefully - the generalized function may have different signatures
- Example: If `original_login_spotify()` becomes `login(app_name='spotify')`, update all calls accordingly
- Preserve all other logic in the single function
**Deduplication:**
- If two functions do exactly the same thing, call the same APIs, keep only one
- If functions are similar but have minor differences, merge them with parameters
**Preservation:**
- Keep all unique functionality. If you believe that they should not be clustered together, then keep both functions
and indicate that in the docstring
- Don't lose edge case handling
- Maintain error handling and validation
**Code Reuse:**
- You can and SHOULD call other generalized functions you've created to build more complex functionality
- Example: If you create a generic `login()` function, you can use it in a `get_authenticated_user()` function
- This creates a hierarchy of reusable components
- Only call: (1) Your own generalized functions, (2) AppWorld APIs (apis.*)
- NEVER call the original learned functions - they won't exist in the final code
...
```