Proceedings of the 2026 Winter Simulation Conference
V. Ramamohan, A. Djanatliev, M. Fakhimi, C. Krejci, C. Ruiz Martin, B. S. Onggo, and N. Mustafee, eds.

BILEVEL OPTIMIZATION OF AGENT SKILLS VIA MONTE CARLO TREE SEARCH

Chenyi Huang[1,3], Haoting Zhang[2], Jingxu Xu[2], Zeyu Zheng[2], and Yunduan Lin[3]

[1]Dept. of Mathematics, National University of Singapore, SINGAPORE
[2]Dept. of Industrial Eng. & Operations Research, University of California at Berkeley, CA, USA
[3]Dept. of Decision, Operations and Technology, The Chinese University of Hong Kong, CHINA

ABSTRACT

Agent `skills` are structured collections of instructions, tools, and supporting resources that help large language model (LLM) agents perform particular classes of tasks. Empirical evidence shows that the design of `skills` can materially affect agent task performance, yet systematically optimizing `skills` remains challenging. Since a `skill` comprises multiple structured components, optimization requires jointly determining both the structure of these components and the content each component contains, creating a complex decision space with strong interdependence. We therefore formulate `skill` optimization as a bilevel optimization problem. We propose a framework in which an outer loop employs Monte Carlo Tree Search to determine the `skill` structure, while an inner loop refines the component content within the selected structure. In both loops, we employ LLMs to assist the optimization procedure. We evaluate the framework on an open-source Operations Research Question Answering dataset, and results suggest that it improves agent performance with the optimized `skill`.

1 INTRODUCTION

Large language model (LLM) agents are increasingly deployed to perform complex, multi-step tasks across diverse domains, such as code generation, data processing, and business analysis. A recent trend that enables task specialization in these systems is the agent `skill` (Anthropic 2025): a collection of software artifacts including task instructions, tools, and supporting resources, organized in a structured folder; see Figure 1 for an example. During task execution, LLM agents equipped with a particular `skill` will query and load the components within the `skill` folder to help task completion. A well-designed `skill` provides relevant guidance and supporting resources in a compact and usable form, whereas a poorly designed `skill` may waste context, introduce irrelevant information, misdirect the agent, and in some cases even degrade task performance. Recent benchmark evidence (Li et al. 2026) shows that `skills` can substantially improve performance, but their effects are highly heterogeneous across tasks. These observations suggest the need to develop systematic strategies for improving `skills` to optimize agent performance.

Studying this design problem requires a perspective different from what is typically optimized in conventional software artifacts. Conventional software artifacts deliver value mainly through executable code, of which the behavior is largely explicit once implemented. By contrast, a `skill` affects agent behavior through natural language instructions and auxiliary materials that must be loaded into the context window and interpreted during reasoning. This creates three central challenges for optimization. First, a `skill` is heterogeneous, combining instructions, executable scripts, reference documents, and structured assets. This enlarges the design space and makes it difficult to represent the optimization problem with an explicit set of decision variables. Second, these components are interdependent. Editing a single component can alter the role, necessity, or interpretation of others, and therefore, each edit of the component requires coordinated revisions elsewhere. Third, even when the decision variables are explicitly represented, the feasible set is discrete and combinatorial, constrained by schema requirements, token-budget limits, and structural consistency conditions. Because of these reasons, `skill` optimization becomes challenging not only to solve, but even to formulate as a well-structured optimization problem.

In this work, we propose a framework to formulate and solve the `skill` optimization as a bi-level optimization problem. Our framework is grounded in the official Agent Skills specification (https://agentskills.io/specification). This specification defines both the structure of a `skill` and how its contents are exposed during execution. Structurally, a `skill` is implemented as a directory centered around a required `SKILL.md` file as the general description, together with optional subdirectories such as `scripts/` for executable code, `references/` for supplementary documentation, and `assets/` for static resources such as templates and lookup tables. The `SKILL.md` file itself contains two parts: a YAML frontmatter block with structured metadata, such as the skill name, routing description, compatibility requirements, and approved tools, followed by a Markdown body containing the agent-facing instructions. Operationally, the specification adopts a progressive-disclosure loading model over this organization: at agent startup, only lightweight frontmatter metadata is loaded; upon skill activation, the full `SKILL.md` content is loaded; and files in the optional subdirectories are loaded on demand. This hierarchy motivates us to represent a `skill` as a tuple $S = (\theta, \phi)$, where θ denotes the structure configuration of the `skill`, and ϕ denotes the instantiated content within the structure.

Given this representation, we formulate `skill` optimization as a bilevel problem in which an outer loop searches over structure configurations θ and an inner loop optimizes the associated content ϕ under each fixed structure. The outer-loop problem is naturally sequential and discrete: structural edits are path-dependent, since an early revision can change which later revisions become feasible, desirable, or necessary. Moreover, the quality of a structure choice is only revealed after subsequent content refinement and downstream evaluation. We therefore organize the revision process as a tree and employ Monte Carlo Tree Search (MCTS) to navigate it. MCTS uses delayed evaluation feedback to favor edit paths that lead to strong empirical performance while preserving exploration of alternative revisions. For each proposed structure, the inner loop first constructs aligned initial content under that structure and then refines it through a bounded sequence of attempts. This refinement is not handled by a single generic procedure. Instead, the inner loop dispatches the proposed structure edit to a refinement family matched to the corresponding type of content work. Refinement then proceeds sequentially within the selected family, i.e., each inner loop explicitly depends on the structure selected by the upstream outer loop. After the allowed attempts are completed, the recorded outcomes are ranked using a conservative selection rule. The top-ranked outcome, together with the evaluation signal, is then returned to the outer loop. In this manner, the bilevel design separates structure search from content refinement, provides an explicit attribution of the evaluation feedback during the optimization procedure. Since both the outer structure search and the inner content refinement require optimizing over unstructured variables that can hardly be represented by numerical values, we employ LLMs to implement the optimization procedures (Yang et al. 2023).

2 BACKGROUND AND RELATED WORK

This section reviews two lines of work most relevant to our framework: tree-search methods for LLM agents, and prior work on agent `skills` as reusable capability modules.

2.1 Tree Search Methods for LLM Reasoning and Agent Optimization

Tree-structured search has emerged as a powerful paradigm for improving LLM reasoning and agent decision-making. Yao et al. (2023) proposed Tree of Thoughts, which extends chain-of-thought prompting by enabling deliberate exploration of multiple reasoning paths over coherent text units ("thoughts"), using search strategies such as breadth-first search and depth-first search for lookahead and backtracking. Building on this, Hao et al. (2023) introduced Reasoning via Planning, which repurposes the LLM as both a world model and reasoning agent within an MCTS framework. Zhou et al. (2024) proposed Language Agent Tree Search, the first general framework that unifies LLM reasoning, acting, and planning through Monte Carlo Tree Search, using LLM-powered value functions and self-reflections together with environment feedback to guide exploration. In the domain of mathematical reasoning, MCTSr (Zhang et al. 2024) integrates

MCTS with iterative self-refinement and self-evaluation, while rStar-Math (Guan et al. 2025) demonstrates that small language models can match frontier-model performance on competition mathematics through MCTS-based deep thinking with process reward models. Together, these works show that tree-based search can improve reasoning by allowing the model to explore, evaluate, and revise intermediate candidates. A natural next step is to apply the same idea not only to reasoning traces, but also to agent artifacts themselves.

Most relevant to our work is AFlow (Zhang et al. 2025), which reformulates agentic workflow optimization as a search problem over code-represented workflows and uses MCTS to iteratively refine them through code modification, tree-structured experience, and execution feedback. AFlow demonstrated that MCTS can effectively navigate the discrete, combinatorial space of agent workflow designs, achieving consistent improvements over manually designed baselines. Our framework shares AFlow's use of MCTS for optimizing agent-facing artifacts, but differs in several respects. First, we optimize `skill` packages rather than code-represented workflows, which introduces a heterogeneous design space spanning instructions, scripts, references, and assets. Second, we adopt a bilevel formulation that explicitly separates structure search from content refinement, whereas AFlow operates on a single level workflow representation. Third, our inner loop uses action-family-aware refinement strategies with a conservative selection rule, rather than applying a uniform code-modification procedure.

Overall, this line of work supports our use of tree search, but existing approaches do not directly address the optimization of structured `skill` packages. Moreover, because the evaluation of LLM-generated artifacts is inherently noisy, our framework is also related to the broader simulation optimization literature, where ranking-and-selection ideas are used to compare noisy alternatives in discrete design spaces (Hong and Zhang 2021; Eckman and Henderson 2021; Du et al. 2024; Fu et al. 2026).

2.2 LLM Agent Skills

Related work on agent `skills` can be broadly organized into two complementary strands: one studies how reusable capabilities can improve agent behavior, and the other treats `skills` as portable system components and focuses on their specification, orchestration, evaluation, and security.

Voyager (Wang et al. 2024) introduced an ever-growing `skill` library of executable code for a lifelong learning agent in Minecraft, where `skills` are automatically generated, verified, and stored for future retrieval. Reflexion (Shinn et al. 2023) proposed verbal reinforcement learning, in which agents improve through linguistic self-reflection rather than weight updates, storing experiential feedback in an episodic memory. More recently, SAGE (Wang et al. 2025) uses reinforcement learning to enable agents to build and refine reusable `skill` libraries, demonstrating self-improvement through iterative `skill` acquisition. These works establish the principle that an agent's capabilities can be improved by optimizing a library of reusable modules rather than the underlying model.

In parallel, a growing body of work has treated agent skills as first-class software artifacts. Anthropic introduced the Agent `Skills` specification (Anthropic 2025).

This specification has since been adopted as an open standard for cross-platform `skill` portability. Li et al. (2026) proposed AgentSkillOS, an operating-system-like framework for organizing and orchestrating skills at the ecosystem scale. On the evaluation side, SkillsBench (Li et al. 2026) provides a large-scale benchmark for assessing how well agent skills generalize across diverse tasks, revealing that `skill` effects are highly heterogeneous and that `skill` quality significantly impacts downstream performance. Complementary security analyses (Liu et al. 2026) have documented the prevalence of vulnerabilities and malicious patterns in deployed `skills`, underscoring the importance of rigorous skill validation.

3 METHODOLOGY

3.1 Framework Overview

We now present the overall framework for `skill` optimization. To study this problem systematically, we represent a `skill` as a tuple $S = (\theta, \phi)$. The variable $\theta \in \Theta$ denotes the structure configuration of

the `skill`, that is, the set of components it contains and their organizational arrangement. The variable $\phi \in \Phi(\theta)$ denotes the instantiated content under that structure, including the instructions in `SKILL.md`, the code in executable scripts, the text in reference documents, and the data stored in static assets. The structure space Θ is discrete and combinatorial because it ranges over alternative structure configurations. In contrast, for a fixed structure θ, the space $\Phi(\theta)$ consists of all content instances compatible with that structure. This space is typically extremely large, since the instructions, code, and supporting materials instantiated under the given structure may still vary widely.

For a given seed `skill` S_0, we formulate `skill` optimization as the bilevel problem

$$\max_{\theta \in \Theta} \max_{\phi \in \Phi(\theta)} R_{S_0}(\theta, \phi),$$

subject to structural-validity and token-budget constraints. The objective $R_{S_0}(\theta, \phi)$ is the scalar performance score assigned to candidate (θ, ϕ) by the evaluation procedure associated with S_0. It is seed-dependent because the downstream tasks and assessment criteria depend on the intended function of the seed `skill`. Structure-validity constraints require a candidate to remain a well-formed `skill` under the specification, for example by preserving the required directory organization and maintaining a well-formed `SKILL.md`. Token-budget constraints require the candidate to remain usable within the context window, for example by preventing the active instructions from becoming too long or the overall directory contents from becoming excessively large. This bilevel formulation reflects the hierarchical nature of the problem: the outer level compares candidate structures, while the inner level evaluates each structure through the best content refinement achievable within it, as illustrated in Figure 1.

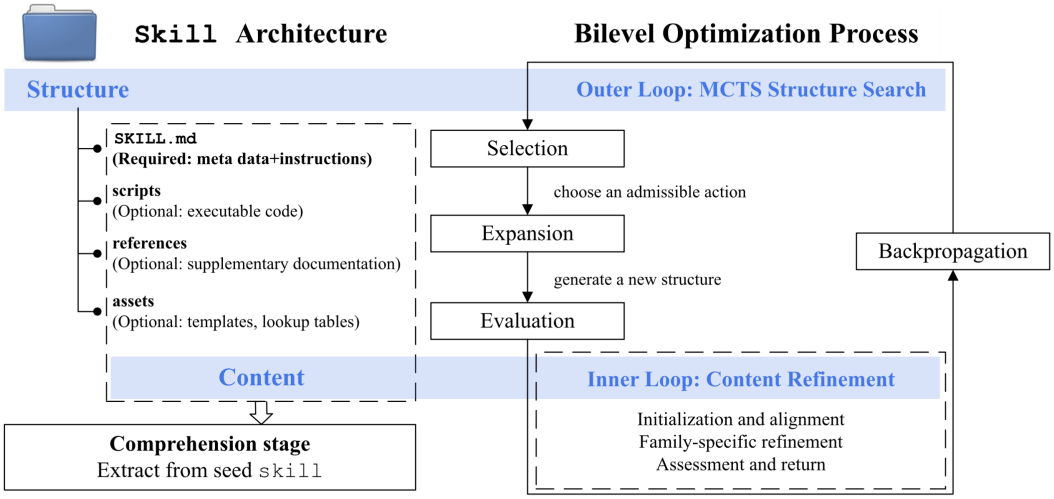

Figure 1: Bilevel Optimization Architecture for Agent Skill Modules.

To address this problem, we develop a framework that begins with a one-time comprehension stage, which converts a raw seed `skill` into an optimization-ready initialization, and then improves the `skill` through bilevel optimization with an outer loop over structure candidates and an inner loop over content refinement.

The **comprehension stage** is a one-time preparation step performed before bilevel optimization begins. Applied to the seed `skill` S_0, it first extracts the files in the directory, infers their structural roles, and parses the frontmatter and section organization of `SKILL.md`, yielding the initial structure representation θ_0. Using θ_0 together with the extracted file contents, it then instantiates the initial aligned content representation ϕ_0. In addition, the comprehension stage constructs a `skill` profile $\mathscr{P}$, which summarizes the task the `skill` is intended to support, the relevant success criteria and quality dimensions, and the

structural revision directions that appear most promising. We use $\mathscr{P}$ to define a profile-guided search region $\widetilde{\Theta}(\mathscr{P}) \subseteq \Theta$. In this way, $\mathscr{P}$ acts as an algorithmic search prior for the outer loop: it narrows the structure search space, guides how structure candidates are explored and provides persistent task-aligned context for exploring candidate revisions. Together, (θ_0, ϕ_0) and $\mathscr{P}$ provide a grounded initialization for the subsequent optimization.

The **bilevel optimization** then starts with the outer level, which searches over alternative structure candidates starting from θ_0. We model this process as a sequential search over a tree of candidate structures, where each node corresponds to a structure state and each edge corresponds to an admissible edit. Because candidate structures are generated, evaluated, and updated iteratively, the search is carried out through an outer loop. The tree-based view reflects the path-dependent nature of structure optimization: an earlier edit can change which later edits become desirable, feasible, or even necessary. For example, adding a detailed instruction section to `SKILL.md` may later require moving supporting details into a separate reference file to remain within the token budget. Guided by the prior $\mathscr{P}$, the outer loop repeatedly selects a promising structure state, proposes a candidate edit, and validates the resulting candidate against structural and budget constraints. Only feasible candidates are passed to the inner level for content refinement and downstream evaluation. The resulting reward and diagnostics are then returned to the outer loop to update the search statistics and guide subsequent exploration. We implement the search at this level using Monte Carlo Tree Search (MCTS), which is well suited to discrete sequential optimization with delayed feedback, since the value of a structure revision is revealed only after content refinement and evaluation at the inner level.

Once the outer loop generates a valid structure candidate, the inner level refines the associated content while keeping the selected structure fixed. Because the outer loop modifies the structure, the system must first transfer the existing content into the new structure through a bridge operation that preserves reusable material while accommodating newly introduced components. Starting from this aligned content state, refinement is carried out through an inner loop. This inner loop performs a bounded number of refinement attempts, with the specific refinement procedure determined by the type of structural edit that triggered it. Each valid refinement outcome is evaluated on the downstream tasks under the current structure, producing a corresponding reward and diagnostic record. These evaluated outcomes are then compared, and one final content is selected for return to the outer loop. Combined with the fixed structure, this selected content determines the candidate `skill` associated with the current structure change.

After multiple rounds of outer-loop search and inner-level refinement, the framework returns the `skill` candidate associated with the highest-reward node in the search tree.

3.2 Outer Loop: MCTS-Based Structure Search

In this subsection, we describe the main components and procedure of the MCTS-based outer loop.

We use n to denote a node in the search tree and $\mathscr{V}$ to denote the current set of tree nodes. The root corresponds to the initial candidate (θ_0, ϕ_0). Each node n stores a structure θ, its associated aligned content state ϕ, and search statistics including the visit count $N(n)$ and the current estimate of its search value $\bar{Q}(n)$. Although both θ and ϕ are stored at each node, the action space of the outer loop operates only on the structure. The content ϕ is retained so that, after a structural revision is proposed, the resulting candidate can be refined by the inner loop and evaluated consistently. The action space $\mathscr{A}$ consists of admissible structure edits, including additions, removals, reorderings, and edits to components such as sections, references, scripts, assets, and metadata. The set of admissible actions varies across nodes, because whether a structural edit can be applied depends on the current structure. Given a current structure θ and an admissible action $a \in \mathscr{A}$, a transition map T defines the successor structure $\theta' = T(\theta, a)$.

The outer loop follows the four-step structure of MCTS: **selection**, **expansion**, **evaluation**, and **backpropagation**. In each iteration, it selects a parent node, proposes and applies a structure edit, evaluates the resulting candidate after inner-loop refinement, and backpropagates the reward to update the tree statistics. In our setting, this basic procedure is adapted in two ways. First, evaluation is bilevel: a structure candidate is evaluated only after its content has been refined by the inner loop and assessed on downstream tasks.

Second, the search is LLM-guided: LLM reasoning is used to analyze the current state, interpret evaluation feedback, and propose the next structural revision.

The MCTS steps begin with **selection**, which chooses an existing node in the current search tree as the parent for the next expansion. The framework supports two selection policies. The default policy is upper confidence bound (UCB), which assigns each selectable node $n \in \mathscr{N}$ the score

$$\mathrm{UCB1}(n) = \bar{Q}(n) + c\sqrt{\frac{\ln \sum_{m \in \mathscr{V}} N(m)}{N(n)}},$$

where $\bar{Q}(n)$ is the empirical mean reward and $c > 0$ is the exploration constant. The framework selects node $n^\star = \arg\max_{n \in \mathscr{N}} \mathrm{UCB1}(n)$. This policy is suitable when the reward signal is relatively stable.

For noisier evaluation settings, the framework also supports a mixed-probability policy that combines uniform exploration with score-based sampling. For each node $n \in \mathscr{N}$, define the centered value $\tilde{Q}(n) = \bar{Q}(n) - \frac{1}{|\mathscr{N}|}\sum_{m \in \mathscr{N}} \bar{Q}(m)$. The sampling distribution is

$$P(n) = \lambda \cdot \frac{1}{|\mathscr{N}|} + (1-\lambda) \cdot \frac{\exp\big(\alpha \tilde{Q}(n)\big)}{\sum_{m \in \mathscr{N}} \exp\big(\alpha \tilde{Q}(m)\big)}, \qquad n \in \mathscr{N},$$

where $\lambda \in [0,1]$ is the mixing coefficient and $\alpha > 0$ is an inverse-temperature parameter. When λ is large, the policy approaches uniform exploration; when λ is small, it places great weight on high-value nodes. This provides a smooth interpolation between broad exploration and focused exploitation, mitigating the risk of premature convergence when reward estimates are highly stochastic.

After a node has been selected, the framework proceeds to **expansion**, which generates a new structure candidate from the selected parent node through a three-stage LLM-guided reasoning procedure. This design is motivated by evidence that decomposing complex tasks into intermediate steps can improve performance (Wei et al. 2022; Zhou et al. 2022). In the *analysis* stage, the system examines the current `skill` structure together with summary evaluation information, the persistent `skill` profile, and specification constraints, aiming to form a structural understanding of the current candidate. In the *diagnosis* stage, this structural understanding is combined with richer evaluation diagnostics and search experience from previous rounds to identify the likely root cause of underperformance and a structural hypothesis for addressing it. In the *proposal* stage, the diagnosis is translated into a concrete action $a \in \mathscr{A}$, conditioned on the available action types, their parameter definitions, and warnings from previous failed or risky edits.

Once an action a is proposed, it is applied to produce $\theta' = T(\theta, a)$. Before θ' is admitted into the tree, the framework applies a structural validation gate to check whether the revised candidate remains a valid `skill` and satisfies the required structural and budget constraints. Candidates that fail these checks are discarded, so that only feasible successors proceed to content refinement.

The next phase is **evaluation**. Because the outer loop modifies only the structure, a newly proposed θ' cannot be evaluated directly. It must first be passed to the inner loop. Under this fixed structure θ', the inner loop constructs aligned content, refines it, and selects one final refined content $\phi^\star(\theta')$. The pair $(\theta', \phi^\star(\theta'))$ then defines the candidate `skill` for the current expansion. This candidate is evaluated on the downstream tasks, and its resulting task performance defines the reward $R_{S_0}(\theta', \phi^\star(\theta'))$ together with the associated diagnostics. The evaluation also produces diagnostic information, which is returned together with the reward for use in subsequent search steps. The details of the inner-loop refinement and evaluation procedure are given in Section 3.3.

The framework then proceeds to **backpropagation**. The reward $R_{S_0}(\theta', \phi^\star(\theta'))$ is propagated along the path from the expanded node back to the root, updating the visit counts and value estimates of the visited nodes. Through these updates, the search tree gradually records which structural revisions tend to yield stronger downstream performance after content refinement.

The four phases together complete one iteration of the outer loop. The search terminates when the iteration budget is exhausted, when no valid expansions remain, or when anti-stagnation controls indicate that further exploration is unlikely to improve performance.

3.3 Inner Loop: Content Refinement

We now turn to the inner loop, which is invoked during the evaluation phase of the outer loop. Once the outer loop proposes a structurally valid candidate θ', the inner loop refines the associated content under that fixed structure. Because the content may include instruction text, reference material, and executable code, this refinement is carried out under a bounded optimization budget and proceeds in three phases: **initialization and alignment**, **family-specific refinement**, and **pessimistic assessment and return**.

The first phase, **initialization and alignment**, introduces the bridge step that connects outer-loop structure search with inner-loop content refinement. When the outer loop proposes a new structure θ', it changes only the organization of the `skill`, rather than the concrete content instantiated under that organization. The candidate is therefore not yet directly executable. To make refinement possible, the framework first constructs aligned initial content $\phi_0(\theta')$ by transferring existing content into the new structure organization, preserving reusable material whenever possible while accommodating any added, removed, or reordered components. This resulting $\phi_0(\theta')$ then provides the initial state for subsequent content refinement.

The second phase is **family-specific refinement.** Different structure changes induce different content-level optimization needs, so the inner loop does not apply a single generic refinement procedure. Instead, each proposed change is dispatched to a refinement family matched to the kind of content work it requires. The framework implements five refinement families: metadata-light updates, metadata-routing text, instruction text, redistribution, and script edit or generation. For example, a change to section organization in `SKILL.md` naturally leads to instruction text refinement, whereas a script-related edit leads to script editing or generation. Once the refinement family has been selected, the inner loop performs a bounded sequence of M refinement attempts within that family. The attempt is the basic sequential unit of refinement. Starting from the aligned content $\phi_0(\theta')$, attempt m takes the current content package as input and returns one refined package as output, which then becomes the input to attempt $m+1$. This produces the trajectory $\phi_0(\theta') \to \phi^{(1)} \to \cdots \to \phi^{(M)}$, where $\phi^{(m)}$ denotes the content returned by attempt m. The process terminates when the budget is exhausted or when further refinement is judged unnecessary.

Each attempt produces an outcome record containing the returned content together with evaluation signals such as its improvement over a comparison baseline, and a confidence measure summarizing the strength of the supporting evidence. In some refinement families, a single attempt may internally compare multiple candidate variants before returning one selected content. When this occurs, and the candidate variants yield improvements $\delta_1, \ldots, \delta_k$ relative to the comparison baseline, the inner loop computes a lower confidence bound $\mathrm{LCB} = \bar{\delta} - t_{\mathrm{crit}} s/\sqrt{k}$, where $\bar{\delta}$ is the mean improvement, s is the sample standard deviation, and t_{crit} is a conservative critical value. This quantity discounts the observed mean improvement by an uncertainty penalty and serves as a pessimistic estimate of gain reliability.

The third phase, **assessment and return**, determines which refinement outcome is passed back to the outer loop. Since downstream evaluation is inherently noisy, a positive observed gain does not by itself guarantee a reliable refinement. The framework therefore uses the lower confidence bound from the second phase as a conservative acceptance criterion: an outcome is treated as having passed the pessimistic gate when its estimated lower confidence bound is nonnegative, that is, when $\mathrm{LCB} \geq 0$. The recorded outcomes are then ranked by first preferring those that pass this pessimistic gate, then those with larger improvement, and finally those with higher confidence. Under this ordering, the top-ranked content package is selected as the accepted content. The accepted content and its diagnostics are returned to the outer loop, which uses the resulting reward for backpropagation. The complete procedure of the `skill` optimization is summarized in Algorithm 1.

4 EXPERIMENTS

In this section, we evaluate the proposed method by optimizing a seed `skill` for the Operations Research Question Answering (ORQA) benchmark dataset (Mostajabdaveh et al. 2025). ORQA is a multiple

Algorithm 1 Bilevel Skill Optimization via MCTS

Require: Seed skill S_0, Maximum search rounds K
Ensure: Optimized skill $S^\star = (\theta^\star, \phi^\star)$

Extract initial structure θ_0, content ϕ_0, and task profile $\mathscr{P}$ from S_0
Initialize search tree $\mathscr{T}$ with root node $n_0 = (\theta_0, \phi_0)$
Initialize best candidate: $\theta^\star \leftarrow \theta_0$, $\phi^\star \leftarrow \phi_0$, and best reward $r^\star \leftarrow -\infty$
for $t = 1, 2, \ldots, K$ **do**
    Select a parent node $n \in \mathscr{T}$ containing structure θ_n and content ϕ_n
    Propose structure-editing action a via LLM conditioned on θ_n and $\mathscr{P}$
    if action a is structurally valid and meets budget constraints **then**
        $\theta' \leftarrow \text{ApplyAction}(\theta_n, a)$ ▷ Outer loop: create new structure
        $\phi'_0 \leftarrow \text{AlignContent}(\phi_n, \theta', a)$ ▷ Inner loop: bridge existing content
        $\phi' \leftarrow \text{RefineContent}(\phi'_0, \theta', a)$ ▷ Inner loop: perform bounded refinement
        $r' \leftarrow R_{S_0}(\theta', \phi')$ ▷ Evaluate downstream task performance
        Add new node $n' = (\theta', \phi')$ to $\mathscr{T}$ and backpropagate r'
        if $r' > r^\star$ **then**
            $\theta^\star \leftarrow \theta'$; $\phi^\star \leftarrow \phi'$; $r^\star \leftarrow r'$ ▷ Update optimal reward
        end if
    end if
    if search convergence criteria are met **then**
        break ▷ Terminate early if search stagnates
    end if
end for
return $S^\star = (\theta^\star, \phi^\star)$

choice question answering task in operations research, in which the agent receives a natural language problem description together with answer options and must infer the underlying optimization model in order to choose the correct answer. Typical questions involve identifying decision variables, constraints, parameters, objectives, or the meaning of modeled expressions. The full ORQA benchmark contains 1,513 instances across 20 application domains, with an average input length of 231 words. Table 1 presents two representative ORQA examples.

Table 1: Representative ORQA examples.

Context (abridged)	**Question**	**Options**	**Ans.**
A broadcast network must choose which shows to air on a single channel. Each show has a duration, a deadline, and a viewer-rating score. Some shows may be omitted if they conflict with others. The goal is to maximize total viewer rating while respecting scheduling constraints.	What are the decision activities of the optimization problem?	(A) Due date (B) Show broadcast order (C) Show broadcast indicator (D) Processing time	(C)
A logistics manager must plan the movement of trucks so that all deliveries are completed using as few vehicles as possible. The data include locations, travel times, delivery requirements, and initial vehicle availability at each location.	Which of the following options are participating decision activities in the objective criterion for this problem?	(A) Number of trucks departing daily from each location (B) Number of deliveries to be made from each location (C) Whether to set a location to never be an origin (D) Travel time between each location	(A)

The seed `skill` used in the ORQA experiment is a small two-file `skill` package generated by `skill-creator`, a built-in `skill` for creating and modifying other `skills`. Our aim is to further optimize this AI-generated seed `skill` for answering ORQA questions. It consists of a main `SKILL.md` file and a supporting reference file on question types. The main `SKILL.md` file provides the agent a basic answering workflow: restate the optimization goal, identify the type of model component being queried, rule out clearly incompatible options, relate the remaining choices back to the problem description, and return exactly one final answer token. It also provides a few heuristics and final checks. The supporting reference file summarizes common ORQA question types and the distinctions among objectives, constraints, variables, parameters, and indexed sets. This seed `skill` serves both as the starting point for optimization and as the baseline for comparison. During optimization, candidate `skills` must satisfy the structural-validity constraints of the Agent Skills specification and remain within the size budget. In particular, the active `SKILL.md` content is constrained by an activation budget of 5000 tokens, with a warning threshold at 3500 tokens; reference files are loaded only on demand, and script code itself is not directly loaded into the model context.

4.1 Experiment Setup

In this subsection, we describe the experiment procedure and parameter settings used to evaluate the proposed bilevel optimization framework on ORQA.

To support bilevel optimization, we partition the original ORQA dataset into *search*, *confirm*, and *test*, whose roles parallel the standard training, validation, and test sets commonly used in machine learning literature. The *search* split is used during optimization to evaluate candidate `skills` and guide the search process. The *confirm* split is used after optimization for model selection across alternative search settings. Finally, the *test* split is held out until the end and used exclusively for the final evaluation of the selected `skill`. To keep the optimization cost manageable, each run uses sampled subsets from these splits; in the reference run, the total sample size is 120 questions.

To ensure consistency, we fix the main LLM and agent settings throughout the experiment. The LLM agents operate in two roles: runtime evaluation, where candidate `skills` are executed in a Harbor sandbox using `openai/gpt-5.2-codex`; and optimization, where the structure search is orchestrated by `openai/gpt-5.4` through DSPy. Token limits are allocated strategically across five pipeline stages as per-call maximums. Specifically, the initial comprehension stage uses 1024 tokens. Within the outer loop, the analysis, diagnosis, and proposal stages are capped at 1536, 1024, and 20000 tokens, respectively. In the inner loop, each content-refinement subroutine is restricted to 1024 tokens. The significantly larger budget reserved for the proposal stage accommodates the complex task of synthesizing the current structural state, evaluation feedback, and search history to formulate the next revision. This strategic allocation of varying computational budgets across different pipeline stages shares conceptual similarities with multi-fidelity simulation optimization, where expensive, high-fidelity evaluations are strictly reserved for the most critical decision-making points (Rhodes-Leader et al. 2018).

Based on these settings, we then compare two search configurations, each corresponding to one fixed set of MCTS hyperparameters. Configuration A is the more conservative setting, using fewer search rounds, deterministic UCB1 selection, earlier convergence, and priority-action-whitelist enforcement. Configuration B is the more exploratory setting, using more search rounds, mixed-probability selection, more patient convergence settings, and a less restricted action space. Table 2 summarizes the main search, convergence, and selection-policy settings. Each configuration produces one optimal candidate `skill` on the *search* split. These configuration-level candidate `skills` are then re-evaluated on the *confirm* split, and the final winner is the one with the highest score on the *confirm* split. Ultimately, this selected winner and the initial seed `skill` are evaluated once on the *test* split to measure final performance improvements.

Table 2: Sweep configurations used in the ORQA experiment.

Setting	Configuration A	Configuration B
Max rounds	3	6
Selection policy	UCB1	Mixed-probability
Exploration constant	1.2	–
α	–	0.55
λ	–	0.25
Min rounds before convergence	2	3
Consecutive stale rounds to stop	2	3
Improvement threshold	0.001	0.001

4.2 Runtime Cost

We report the runtime cost of the ORQA experiment to make the practical overhead of the framework clearer. As described above, the LLM agents serve two roles: runtime evaluation, where candidate `skills` are executed in a Harbor sandbox; and optimization, where the structure search is orchestrated through DSPy. In this run, the full experiment involved 752 Codex agent executions, consumed 45.95M tokens as recorded in the execution logs, and had an end-to-end wall-clock span of approximately 8.0 hours. The evaluation side accounts for most of the measured runtime: the Harbor logs record 6.62 hours of accumulated Codex-agent execution time, corresponding to repeated executions used to search over, confirm, test, and compare candidate skills. By contrast, the optimization process itself was much smaller, requiring only 29 outer-loop LLM calls and approximately 50 inner-loop DSPy calls. These numbers show that the dominant practical cost comes from repeatedly executing and evaluating candidate skills, rather than from generating or editing the skills themselves.

4.3 Results

In this subsection, we report the experiment results.

During the optimization procedure, both Configuration A and Configuration B reached the same peak reward of 0.9434 on *search* split. The deciding difference emerged at the confirmation stage: Configuration A achieved a mean score of 0.8571 on the sampled *confirm* split, whereas Configuration B achieved 0.8857. Consequently, the winning skill was selected from Configuration B. When evaluated on the held-out *test* split, the seed `skill` established a baseline exact-match score of 0.90625. The final optimized `skill` from Configuration B achieved a score of 0.9375, yielding an overall improvement of $+0.03125$.

Figure 2 illustrates the actual MCTS search tree produced by Configuration B. The figure clearly delineates the winning path, demonstrating that the outer loop explored several structure alternatives before finalizing the `skill`. Starting from the seed `skill`, the search first promoted a composite edit that revised the `skill` description and relocated the key question-type guidance from the reference file directly into `SKILL.md`. Subsequently, it added a dedicated `Question-Type Triage Checklist` section, which yielded the best reward in the tree on *search* split. The figure also displays branches that were explored but ultimately abandoned. This highlights that the search algorithm actively compared multiple structural options rather than defaulting to a single, fixed revision path.

Next, we provide a comparative analysis of the winning `skill` and the seed `skill`, focusing on their concrete differences and the underlying drivers of the observed performance improvements.

At the structure level, the seed skill has two files: a main `SKILL.md` file and a separate reference file on question types. By contrast, the winning skill keeps only a single `SKILL.md` file. The key question-type guidance that was previously stored in the separate reference file is moved into the main instruction file, and the winning skill also adds a dedicated `Question-Type Triage Checklist` section. As a result, the skill becomes more self-contained and easier to follow. This centralization likely drives performance improvements, as the agent can now process critical classification guidance directly within its primary instruction surface rather than navigating distributed files.

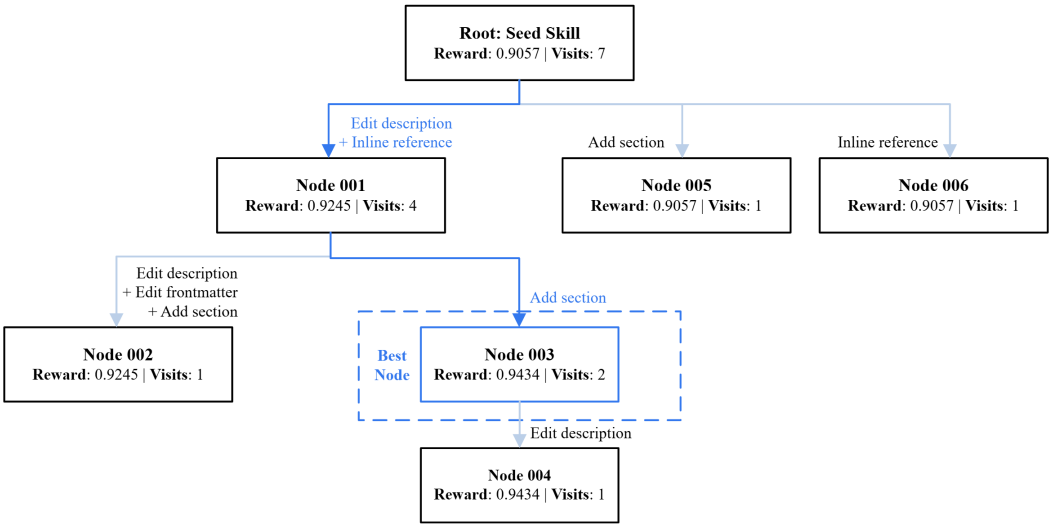

Figure 2: Illustration of MCTS search tree for Configuration B in the ORQA experiment. Nodes are labeled by the structure action and the resulting reward on *search* split. The winning path and selected node are highlighted in blue, while weaker alternatives are shown in a faded style.

At the content level, the winning SKILL.md makes the agent's expected actions more explicit by turning broad guidance into concrete execution rules. For example, the seed skill only states "Classify the question before solving," followed by broad categories such as objectives, constraints, variables, and parameters. The optimized skill instead specifies when this step must occur: "Use this checklist immediately after validating the Input Contract and before any variable, constraint, or objective mapping." It further clarifies how the classification should control later reasoning, adding the rule: "If multiple-choice but non-modeling OR: do not force variable/constraint mapping." In addition, the optimized skill strengthens the final verification stage by requiring the agent to re-check the answer format, verify that the selected option is directly supported by the context, and eliminate every alternative before producing the final single-token answer.

Taken together, the comparison suggests that the overall performance gain stems from synergistic improvements in both structure and content. The outer loop successfully reorganizes the skill, migrating essential guidance into a highly accessible structural format, while the inner loop contributes by making that guidance clearer, more explicit, and more constrained.

REFERENCES

Anthropic 2025, October. "Equipping Agents for the Real World with Agent Skills".

Du, J., S. Gao, and C.-H. Chen. 2024. "A contextual ranking and selection method for personalized medicine". *Manufacturing & Service Operations Management* 26(1):167–181 https://doi.org/10.1287/msom.2022.0232.

Eckman, D. J., and S. G. Henderson. 2021. "Fixed-confidence, fixed-tolerance guarantees for ranking-and-selection procedures". *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 31(2):1–33 https://doi.org/10.1145/3432754.

Fu, M. C., J. Hu, and K. Scheinberg. 2026. "Stochastic Gradients: Optimization, Simulation, Randomization, and Sensitivity Analysis". *IISE Transactions* 58(2):240–256 https://doi.org/10.1080/24725854.2025.2469839.

Guan, X., L. L. Zhang, Y. Liu, N. Shang, Y. Sun, Y. Zhu, *et al*. 2025. "rStar-Math: Small LLMs Can Master Math Reasoning with Self-Evolved Deep Thinking". In *Proceedings of the 42nd International Conference on Machine Learning*, Volume 267 of *Proceedings of Machine Learning Research*, 20640–20661: PMLR.

Hao, S., Y. Gu, H. Ma, J. J. Hong, Z. Wang, D. Z. Wang *et al*. 2023. "Reasoning with Language Model is Planning with World Model". In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* https://doi.org/10.18653/v1/2023.emnlp-main.507.

Hong, L. J., and X. Zhang. 2021. "Surrogate-based simulation optimization". In *Tutorials in Operations Research: Emerging Optimization Methods and Modeling Techniques with Applications*, 287–311. INFORMS https://doi.org/10.1287/educ.2021.0225.

Li, H., C. Mu, J. Chen, S. Ren, Z. Cui, Y. Zhang, *et al*. 2026. "Organizing, Orchestrating, and Benchmarking Agent Skills at Ecosystem Scale". *arXiv preprint arXiv:2603.02176*.

Li, X., W. Chen, Y. Liu, S. Zheng, X. Chen, Y. He, *et al*. 2026. "SkillsBench: Benchmarking how well agent skills work across diverse tasks". *arXiv preprint arXiv:2602.12670*.

Liu, Y., W. Wang, R. Feng, Y. Zhang, G. Xu, G. Deng, *et al*. 2026. "Agent Skills in the Wild: An Empirical Study of Security Vulnerabilities at Scale". *arXiv preprint arXiv:2601.10338*.

Mostajabdaveh, M., T. T. L. Yu, S. C. B. Dash, R. Ramamonjison, J. S. Byusa, G. Carenini, *et al*. 2025. "Evaluating llm reasoning in the operations research domain with orqa". In *Proceedings of the AAAI Conference on Artificial Intelligence*, Volume 39, 24902–24910 https://doi.org/10.1609/aaai.v39i23.34673.

Rhodes-Leader, L., D. J. Worthington, B. L. Nelson, and B. S. Onggo. 2018. "Multi-fidelity simulation optimisation for airline disruption management". In *2018 Winter Simulation Conference (WSC)*, 2179–2190. IEEE https://doi.org/10.1109/WSC.2018.8632329.

Shinn, N., F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. 2023. "Reflexion: Language Agents with Verbal Reinforcement Learning". In *Advances in Neural Information Processing Systems*, Volume 36 https://doi.org/10.52202/075280-0377.

Wang, G., Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, *et al*. 2024. "Voyager: An Open-Ended Embodied Agent with Large Language Models". *Transactions on Machine Learning Research*.

Wang, J., Q. Yan, Y. Wang, Y. Tian, S. S. Mishra, Z. Xu, *et al*. 2025. "Reinforcement Learning for Self-Improving Agent with Skill Library". *arXiv preprint arXiv:2512.17102*.

Wei, J., X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, *et al*. 2022. "Chain-of-thought prompting elicits reasoning in large language models". *Advances in neural information processing systems* 35:24824–24837.

Yang, C., X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou *et al*. 2023. "Large language models as optimizers". In *The Twelfth International Conference on Learning Representations*.

Yao, S., D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao *et al*. 2023. "Tree of Thoughts: Deliberate Problem Solving with Large Language Models". In *Advances in Neural Information Processing Systems*, Volume 36, 11809–11822.

Zhang, D., X. Huang, D. Zhou, Y. Li, and W. Ouyang. 2024. "Accessing GPT-4 Level Mathematical Olympiad Solutions via Monte Carlo Tree Self-refine with LLaMa-3 8B". *arXiv preprint arXiv:2406.07394*.

Zhang, J., J. Xiang, Z. Yu, F. Teng, X. Chen, J. Chen, *et al*. 2025. "AFlow: Automating Agentic Workflow Generation". In *The Thirteenth International Conference on Learning Representations*. Oral presentation.

Zhou, A., K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. 2024. "Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models". In *Proceedings of the 41st International Conference on Machine Learning*, Volume 235 of *Proceedings of Machine Learning Research*, 62138–62160.

Zhou, D., N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, *et al*. 2022. "Least-to-most prompting enables complex reasoning in large language models". *arXiv preprint arXiv:2205.10625*.

AUTHOR BIOGRAPHIES

CHENYI HUANG is a graduate student at the National University of Singapore. She also serves as a Research Assistant at The Chinese University of Hong Kong. Her e-mail address is e1349254@u.nus.edu.

HAOTING ZHANG received his Ph.D. degree in the Department of Industrial Engineering and Operations Research at the University of California, Berkeley. His research interests include stochastic modeling/simulation and artificial intelligence. His e-mail address is haoting_zhang@berkeley.edu.

JINGXU XU received his Ph.D. degree in the Department of Industrial Engineering and Operations Research at the University of California, Berkeley. He has done research in simulation, stochastic modeling, and data analytics. His email address is jingxu_xu@berkeley.edu.

ZEYU ZHENG is an Associate Professor at the University of California at Berkeley in the Department of Industrial Engineering and Operations Research and the Berkeley Artificial Intelligence Research Lab (BAIR). His email address is zyzheng@berkeley.edu and his website is https://zheng80.github.io/.

YUNDUAN LIN is an Assistant Professor in the Department of Decisions, Operations, and Technology at the Chinese University of Hong Kong. Her email address is yunduanlin@cuhk.edu.hk and her website is https://www.bschool.cuhk.edu.hk/staff/lin-yunduan/.