# OR-SKILL: SELF EVOLUTIONARY AGENT SKILLS FOR OPTIMIZATION MODELING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Optimization modeling is crucial for decision-making. Constructing optimization models from natural language descriptions has long been a challenging research goal. Prior work has predominantly relied on fine-tuned large language models (LLMs) or LLM-based agentic workflows to automate this process. However, the rise of general-purpose coding agents, such as Claude Code, offers a new paradigm for tackling such complex tasks. In this paper, we present OR-Skill, a framework that equips a general-purpose agent with a specialized skill for optimization modeling, where the skill is evolved iteratively in an environment built from our curated benchmark. The evolution process operates in an iterative loop. In each iteration, the agent executes tasks with a candidate skill, receives feedback, attributes failures to their root causes, and summarizes them into consolidated diagnostic reports at different error levels. Based on these reports, we perform cascade evolution in reverse order of error severity to refine the skill. We conduct extensive experiments across seven evaluation subsets from six optimization-modeling benchmark suites, on which OR-Skill achieves state-of-the-art or comparable performance. Compared with the strongest baselines, OR-Skill improves accuracy by $4.8$ percentage points on IndustryOR and $8.9$ points on ComplexLP.

## 1 INTRODUCTION

Optimization modeling is central to real-world decision-making in domains including healthcare (Pierskalla & Brailer, 1994), production planning (Bitran et al., 1986), and public service management (Toregas et al., 1971). Traditionally, constructing an optimization model requires substantial domain expertise and manual effort, making it labor-intensive and error-prone. Recent advances in large language models (LLMs) have opened new opportunities for automating this process. Existing studies have explored generating optimization models directly from natural language descriptions, often relying on fine-tuned open-source LLMs (Huang et al., 2024a) or task-specific agentic workflows with hand-designed role pipelines (Xiao et al., 2024; AhmadiTeshnizi et al., 2024).

However, the emergence of general-purpose coding agents equipped with specialized skills, such as Claude Code[1], has shifted the research paradigm away from manually crafted workflow design. Unlike fixed workflows, general-purpose agents can autonomously decompose goals, reason step by step, invoke tools, and reflect on failed attempts as needed. Meanwhile, agent *skills* have been proposed as task-specific operating policies that bundle workflow instructions, domain-knowledge references, and executable scripts (Xu & Yan, 2026).

Figure 1 shows a product-planning problem from an industrial scenario. For OptiMUS (AhmadiTeshnizi et al., 2024), the LLM follows a fixed formulate/code/debug pipeline for every problem. This rigid process prevents the LLM from adapting its analysis strategy to such a hard problem and leaves the production variables unbounded. In contrast, the general-purpose coding agent writes a quick heuristic, identifies which variables dominate the solution, tightens their bounds, and successfully solves it in seconds, which is a smarter strategy. However, the solution is still suboptimal because the agent treats unmet demand as lost sales instead of carrying it over to the next period. These two cases indicate that a general-purpose agent can be a strong modeler, but only when it is equipped with sufficient domain knowledge.

[1] https://github.com/anthropics/claude-code



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

In this paper, we propose OR-Skill, a framework that integrates agent *skills* with a general-purpose agent for optimization modeling. Since manually curating a comprehensive optimization-modeling skill requires substantial OR expertise, OR-Skill adopts the idea of self-evolution, applying evolutionary algorithms on top of LLMs to iteratively improve a skill within an environment. Our framework addresses three key questions: (i) what should be evolved for a skill? (ii) how to construct an environment that provides reliable feedback on modeling results? and (iii) how can the skill be effectively improved in each iteration based on this feedback?

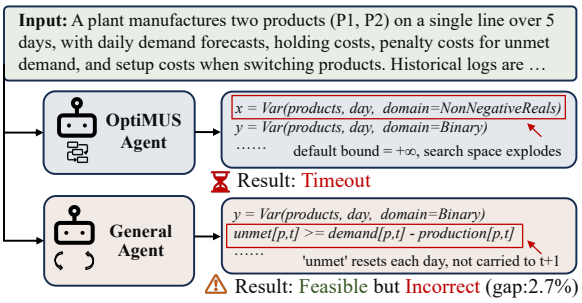



Figure 1: OptiMUS times out from loose variable bounds; a general-purpose agent solves it but misses the backlog formulation, landing 2.7% off optimal.

For the first question, we represent the agent skill as a structured package consisting of a top-level `SKILL.md` file, references that contain domain knowledge and few-shot examples, and executable scripts. The agent accesses these components on demand through progressive disclosure (Xu & Yan, 2026). We version each skill as a Git repository, where every evolution step corresponds to a commit. This design makes the skill easy to inspect, modify, and manage throughout self-evolution.

Second, to provide useful feedback for evolution, we manually construct Hard100, an evolution environment consisting of 100 challenging optimization modeling problems. Each instance corresponds to a representative problem type sourced from industrial scenarios and is carefully annotated by human experts. The annotations include both the optimal objective value and the ground-truth model. The optimization-modeling skill is then evolved according to the agent's performance on Hard100.

For the third question, we design a pipeline to effectively evolve the skill. For each instance in Hard100, we run the agent with the current skill and send its output, together with the ground-truth model, to an LLM-based evaluator. The result is classified into four levels: $L_0$ (Correct), $L_1$ (Syntax Error), $L_2$ (Improvable), and $L_3$ (Logical Error). For each non-correct level, we further attribute failures to their root causes and distill them into a consolidated diagnostic report, producing three level-wise reports per round. Based on these reports, a cascade evolution procedure updates the skill in reverse order of error severity. Empirically, this ordering matters: applying simpler fixes later prevents the skill from overspecializing to hard cases while preserving its basic competence.

OR-Skill obtains excellent results on seven evaluation subsets from six benchmark suites. On the challenging benchmarks, it improves over EvoSkill, the strongest baseline on each: 78.6% versus 73.8% on IndustryOR, 91.0% versus 82.1% on ComplexLP, and 71.4% versus 69.1% on ComplexOR, yielding gains of 4.8, 8.9, and 2.3 percentage points, respectively. These gains suggest that self-evolved skills empower general-purpose agents to handle complex optimization modeling problems that prior methods struggle with. Our contributions are summarized as follows:

- We introduce a domain-specific framework that equips general-purpose agents with self-evolved optimization-modeling skills.
- We develop a report-driven skill evolution pipeline with level-wise grading, root-cause clustering, cascade improvement, and Occam-style length regularization.
- We release the agent skill for optimization modeling, along with its self-evolution framework and dataset Hard100, to support community use and future research.
- The superiority of our proposed method's experimental result suggests a promising direction in optimization modeling beyond training-based methods.

## 2 Related Work

**LLM agents.** Research on LLM agents spans prompting, agentic workflows, and general-purpose agents. Chain-of-Thought prompting (Wei et al., 2022) shows that explicit intermediate reasoning steps can improve performance on complex tasks, while Tree-of-Thoughts (Yao et al., 2023) and

Graph-of-Thoughts (Besta et al., 2024) further organize reasoning as structured search over intermediate thoughts. Another line of work builds agentic workflows, where LLMs follow predefined procedures for planning, revision, or multi-agent collaboration (Wu et al., 2024; Hong et al., 2024). General-purpose agents such as Claude Code and OpenClaw interleave tool calls with execution feedback. ReAct (Yao et al., 2022) couples reasoning with actions; Toolformer (Schick et al., 2023) studies how LLMs can learn to use external tools; and Reflexion (Shinn et al., 2023) uses external feedback to improve future behavior. Our work follows this general-agent paradigm and focuses on equipping a general-purpose agent with a domain skill for optimization modeling.

**LLMs for optimization modeling.** Optimization modeling translates natural language descriptions into solver-ready mathematical models (Ramamonjison et al., 2023). Existing methods fall into training-based and agent-based approaches. Training-based methods improve open-source LLMs through supervised fine-tuning on synthetic data (Huang et al., 2024b; Yang et al., 2025b), as in ORLM (Huang et al., 2024a), OptMATH (Lu et al., 2025), and Step-Opt-Instruct (Wu et al., 2025), or through reinforcement learning, as in SIRL (Chen et al., 2025), DeepOR (Xiao et al., 2026), OR-R1 (Ding et al., 2026), and MURKA (Xie et al., 2025). In parallel, agent-based methods improve optimization modeling by using prompting strategies or multi-agent workflows. CAFA (haoxuan deng et al., 2024), Autoformulation (Astorga et al., 2025) and OptiTree (Liu et al., 2025) use structured prompting, while Chain-of-Experts (Xiao et al., 2024) and OptiMUS (AhmadiTeshnizi et al., 2024) build collaborative agent workflows to mimic the modeling process of human experts. More recently, OptSkills (Yang et al., 2026) distills reusable workflow skills from solution trajectories grouped by problem archetype and refines or expands the library through further learning. OR-Skill instead organizes skill updates around level-wise root-cause reports and cascade evolution.

**Self-evolution.** Self-evolution methods differ in what they update. OPRO (Yang et al., 2023), Promptbreeder (Fernando et al., 2024), and TextGrad (Yüksekgönül et al., 2024) optimize prompts based on task feedback. Beyond prompt-level optimization, Dynamic Cheatsheet (Suzgun et al., 2025) and ACE (Zhang et al., 2025) update a context document after each execution. Other methods evolve reusable tools, with Alita (Qiu et al., 2025) and Voyager (Wang et al., 2023) accumulating tool libraries from successful trajectories. At the level of code, algorithms, and agent architectures, AlphaEvolve (Novikov et al., 2025) and ReGenesis (Peng et al., 2025) apply self-evolution to algorithm discovery and continual reasoning improvement, while ADAS (Hu et al., 2025) lets a meta-agent invent new agent designs in code space. Agent Symbolic Learning (Zhou et al., 2024) further optimizes prompts, tools, and pipelines jointly as symbolic weights. EvoSkill (Alzubi et al., 2026) evolves reusable agent skills through failure analysis and validation-based selection. OR-Skill focuses on optimization modeling, combining level-wise diagnosis, cascade updates, and length-regularized selection.

# 3 METHOD

## 3.1 PROBLEM SETUP

Consider the task of translating natural language descriptions into optimization models. Given a problem description $P_i$, the agent is supposed to produce an executable program $\widehat{M}_i$ that contains the correct decision variables, constraints, and objective. Let $M_i$ denote the expert-annotated ground-truth model for $P_i$. We take the agent skill $s$, a self-contained bundle of domain knowledge, workflow instructions, and executable scripts integrated with a general-purpose coding agent, as the direct optimization target. Formally, we optimize:

$$s^* \;=\; \arg\max_{s}\; \mathbb{E}_{P_i\sim\mathcal{D}}\big[\,\mathrm{Correct}\big(\widehat{M}_i(s),\, M_i\big)\,\big], \tag{1}$$

where $\widehat{M}_i(s)$ is the model produced by the agent equipped with skill $s$ on problem $P_i$, and $\mathcal{D}$ is an environment of annotated problems that provides feedback to drive evolution. The indicator $\mathrm{Correct}$ is 1 when the evaluator judges the generated model correct and 0 otherwise. It is distinct from the length-regularized skill score in Equation 3.

## 3.2 METHOD OVERVIEW

Figure 2 illustrates the full pipeline. OR-Skill operates in a loop. At the beginning of each iteration, a skill $s_t$ is sampled from the database by a performance-weighted sampler, where higher-scoring skills

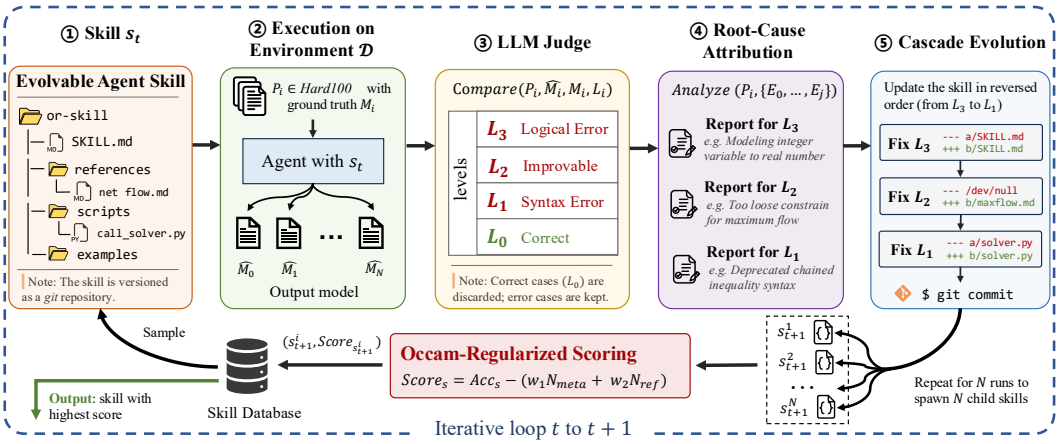



Figure 2: Overview of the OR-Skill self-evolution loop. In each round, the current skill $s_t$ is executed on the environment Hard100, graded by an LLM judge into four levels, attributed to root causes, and then updated via cascade evolution. The resulting candidate skills are scored with an Occam-regularized objective and stored in a skill database, from which the next parent is sampled.

are more likely to be selected. The sampled skill is then integrated with the agent and executed on every problem in the environment $\mathcal{D}$, producing candidate models $\{\widehat{M}_i\}$. An LLM judge compares each generated model $\widehat{M}_i$ against the ground-truth model $M_i$ and assigns it to one of four levels from $L_0$ to $L_3$. Error cases are then analyzed by a root-cause attribution module, which produces one consolidated diagnostic report per level. Based on these reports, the evolution step uses an LLM to update the skill through Git commits. This process is repeated $N$ times per round, producing $N$ child skills. Each child skill is scored by an objective function before being inserted into the skill database. The next skill is then sampled from the database by the same performance-weighted sampler. After $T$ iterations, the skill with the highest score is returned as the final skill $s^*$.

### 3.3 Skill Structure

Formally, the agent's behavior in modeling is conditioned on a skill $s$, as defined in Equation 2.

$$\widehat{M}_i = \mathcal{A}_\theta(P_i; s), \qquad s = (m, \mathcal{R}, \mathcal{E}), \tag{2}$$

Here, $\mathcal{A}_\theta$ denotes the agent, $\widehat{M}_i$ is the generated model for problem $P_i$, $m$ denotes the skill description, $\mathcal{R}$ is a set of reference files, and $\mathcal{E}$ is a set of executable scripts. Here, the reference folder contains Markdown files with domain knowledge for specific classes of optimization problems (e.g. maximum-flow problems) as well as few-shot examples that can be retrieved by the agent when needed. The executable scripts, such as code that invoke solvers to validate the generated model.

Each evolution step modifies the skill through a Git commit, yielding an atomic change to the repository. We initialize the skill $s_0$ with a minimal meta prompt written in `SKILL.md`, without any manually curated domain references or executable scripts.

### 3.4 Environment: Hard100

Self-evolution requires an environment that can provide verifiable and information-dense feedback. To this end, we construct Hard100, a hand-curated collection of 100 challenging optimization-modeling problems covering typical problem types. Each instance contains (i) a natural-language description $P_i$, (ii) an expert-written reference implementation $M_i$, and (iii) the final optimal objective value. The problems span production planning, scheduling, network flow, and other classical OR domains, and each is designed to test at least one non-trivial modeling subtlety, such as tight bounding, integrality, backlog, or piecewise costs. To prevent leakage, Hard100 is written from scratch and is not drawn from any public benchmark used at evaluation time.

### 3.5 Error Judgment

For every generated model, we run an LLM judge that takes the tuple $(P_i, \widehat{M}_i, M_i)$, together with the execution trace and solver outcome, and classifies the result into one of four categories: $L_0$ **(Correct)**: the solver outcome matches the optimal value; $L_1$ **(Syntax Error)**: the code fails to execute or cannot be solved due to implementation errors; $L_2$ **(Improvable)**: the model runs and is partially correct, but contains debatable modeling choices, such as overly loose constraints or ambiguous assumptions that admit multiple valid formulations; and $L_3$ **(Logical Error)**: the code runs successfully, but the optimal answer is wrong. The LLM judge follows a two-step procedure with the full prompt provided in Appendix A.4. After judgment, only error cases from $L_1$ through $L_3$ are retained for downstream modules, while $L_0$ cases are discarded.

### 3.6 Root-Cause Attribution

Feeding dozens of judgment results directly to the LLM is not a good choice. It is expensive and counterproductive because it encourages the LLM to memorize instance-specific symptoms rather than learn transferable modeling knowledge. To prevent this, we introduce a root-cause attribution module. For each level $\ell \in \{L_1, L_2, L_3\}$, the module receives the problems $\{P_i\}$ and the per-instance error messages $\{E_0, \ldots, E_j\}$ at that level and produces a single consolidated report.

The prompt is detailed in Appendix A.4. The design principle behind the attribution prompt is to force the model to reason at the cause level rather than the symptom level. Specifically, the prompt asks the model to cluster errors by their underlying causes, identify the missing capability or misconception behind each cluster, state each finding as a rule for unseen problems in the same family, and reject summaries that hard-code numerical answers or single-instance details.

### 3.7 Cascade Evolution

Given the three diagnostic reports, the LLM updates the skill one level at a time. Counterintuitively, we find it more effective to apply updates in reverse order of error severity, i.e., $L_3 \to L_2 \to L_1$. One might expect the hardest fixes to be applied last, so that the skill improves gradually from simple to difficult cases. However, in practice, evolving from easy to hard causes the final commits to be dominated by a few difficult instances, which can lead the skill to overfit hard cases and degrade general performance. In contrast, evolving from hard to easy applies hard-case fixes early and mitigates this problem, while later easy-case passes serve as sanity checks that restore the skill's basic competencies before the round ends.

Each evolution step modifies the skill through a Git commit. Repeating the full cascade $N$ times yields $N$ child skills $\{s_{t+1}^1, \ldots, s_{t+1}^N\}$, providing the diversity needed for selection. Since the LLM temperature is nonzero, the child skills can differ from one another.

### 3.8 Occam-Regularized Selection

Another failure mode of evolutionary algorithms is unbounded growth. In each round, the LLM may commit a few additional rules, causing the skill to become increasingly longer. Eventually, the marginal gains can be outweighed by context bloat and over-specification. To mitigate this issue, we introduce an Occam-style penalty on skill length into the objective function. For a candidate skill $s$, we define the score as shown in Equation 3.

$$\mathrm{Score}(s) \;=\; \mathrm{Acc}(s) \;-\; \big(w_1 N_{\mathrm{meta}}(s) + w_2 N_{\mathrm{ref}}(s)\big), \tag{3}$$

Here, $\mathrm{Acc}(s)$ is the accuracy on Hard100, $N_{\mathrm{meta}}(s)$ and $N_{\mathrm{ref}}(s)$ denote the token lengths of skill description and all reference files, respectively, and $w_1, w_2 > 0$ are penalty weights. This explicit length penalty encourages the skill to contain concise and general rules.

### 3.9 Implementation Details

Algorithm 1 summarizes the evolution loop (lines 3–13). In each round, $N$ child skills are generated from the same sampled parent. Each child skill is forked from its parent into a separate branch via `git worktree` (line 9), isolating execution artifacts while sharing Git history. The repository

**Algorithm 1** OR-Skill: Self-Evolution of Agent Skills

**Input:** initial skill repository $s_0$, environment Hard100, rounds $T$, children per round $N$
Initialize skill database $\mathcal{S} \leftarrow \{(s_0, \text{Score}(s_0))\}$
**for** $t = 0$ to $T - 1$ **do**
  Sample parent $s_t \sim \mathcal{S}$ proportional to score
  Run $s_t$ on Hard100 to obtain $\{\widehat{M}_i\}$
  Grade each $(P_i, \widehat{M}_i, M_i)$ into $\{L_0, L_1, L_2, L_3\}$
  Attribute errors for levels $\{L_1, L_2, L_3\}$
  **for** $k = 1$ to $N$ **do**
    Fork a new branch from $s_t$ via `git worktree`
    Apply cascade edits $L_3 \to L_2 \to L_1$ as Git commits $\Rightarrow s_{t+1}^k$
    Insert $(s_{t+1}^k, \text{Score}(s_{t+1}^k))$ into $\mathcal{S}$
  **end for**
**end for**
**Output:** $s^* = \arg\max_{s \in \mathcal{S}} \text{Score}(s)$

serves as the skill database, storing each candidate's commit SHA, parent SHA, and score. Cascade evolution uses a LIFO stack of unified diffs. At each level, the LLM generates a patch; the system pushes a `pre_L{k}` tag onto the stack before applying the patch with `git apply`.

# 4 Experiments

## 4.1 Experimental Setup

**Benchmarks.** We evaluate OR-Skill on seven subsets from six benchmark suites with varying levels of difficulty. The easier subsets include NL4Opt (Ramamonjison et al., 2023), NLP4LP (AhmadiTeshnizi et al., 2024), ReSocratic (Yang et al., 2025b), and EasyLP from MAMO (Huang et al., 2024b). The harder subsets include ComplexOR (Xiao et al., 2024), IndustryOR (Huang et al., 2024a), and ComplexLP from MAMO. Following prior work (Chen et al., 2025), we treat MAMO as a single benchmark suite and report EasyLP and ComplexLP as two evaluation subsets. We use the cleaned benchmark collection provided by the survey (Xiao et al., 2025), where incorrectly labeled instances are filtered out. Hard100 is used only as the evolution environment, while the above benchmark suites are reserved for final evaluation. To prevent data leakage, we ensure that Hard100 does not contain cases that are highly similar to those in the evaluation datasets.

**Baselines.** We compare OR-Skill with three categories of methods. The first is the standard LLMs calling, which prompts an LLM to generate the optimization model directly. The second category is agent-based methods, including Chain-of-Experts (Xiao et al., 2024), OptiMUS (AhmadiTeshnizi et al., 2024), Autoformulation (Astorga et al., 2025), CAFA (haoxuan deng et al., 2024), and OptiTree (Liu et al., 2025). We also include two agents with self-evolved skills: EvoSkill (Alzubi et al., 2026), which creates and refines skills through failure analysis, and OptSkills (Yang et al., 2026), which distills skills from clustered optimization trajectories. The third category consists of training-based models. Among them, ORLM (Huang et al., 2024a), and OPTMath (Lu et al., 2025) are trained on synthetic data, while SIRL (Chen et al., 2025), DeepOR (Xiao et al., 2026), OR-R1 (Ding et al., 2026), and MURKA (Xie et al., 2025) are trained with reinforcement learning. For fairness, we use DeepSeek V3.2 as the base LLM for the standard prompting baseline, all agent-based baselines, and OR-Skill. For the training-based methods we rerun, we use Qwen3-8B (Yang et al., 2025a) as the base model, since it is a widely adopted open-source model with strong chain-of-thought reasoning ability.

**Evaluation.** We use GPT-5 (OpenAI, 2026) at temperature 0 to compare each generated program's printed output with the ground truth, allowing a tolerance of 0.01. The LLM judge handles different answer formats and multiple solutions. For example, ReSocratic provides decision-variable assignments and optimal values, whereas IndustryOR provides only the final answer.

Table 1: Accuracy on seven evaluation subsets from six benchmark suites. Unstarred results are reproduced on the cleaned evaluation data; agent-based methods use a unified setup. Results marked with * are copied from the original papers and have not been verified under the same data-cleaning and evaluation protocol, so they are included for reference rather than direct comparison.

| Methods | NL4Opt | NLP4LP | ReSocratic | EasyLP | ComplexOR | IndustryOR | ComplexLP |
|---|---|---|---|---|---|---|---|
| Standard | 80.0% | 84.3% | 78.9% | 91.7% | 42.8% | 47.6% | 41.4% |
| ORLM | 85.9% | 76.4% | 61.8% | 90.4% | 50.0% | 41.3% | 59.5% |
| OPTMath | 94.3% | 73.9% | 58.7% | 87.0% | 50.0% | 43.5% | 62.1% |
| SIRL | 96.2% | 80.6% | 72.6% | 91.8% | 53.6% | 45.7% | 63.4% |
| DeepOR | 97.7% | 82.9% | 73.8% | 93.2% | 64.3% | 52.2% | 67.1% |
| OR-R1 | 88.3%* | 84.6%* | 62.9%* | 86.1%* | 46.3%* | 35.3%* | 49.9%* |
| MURKA | 93.5%* | - | - | 95.5%* | - | - | 55.6%* |
| Chain-of-Experts | 59.6% | 78.1% | 44.4% | 71.4% | 39.3% | 47.6% | 42.3% |
| OptiMUS | 75.9% | 79.8% | 84.9% | 87.2% | 32.1% | 57.1% | 45.1% |
| Autoformulation | 92.2% | 81.5% | 81.6% | 93.6% | 50.0% | 52.4% | 67.6% |
| CAFA | 87.0% | 91.6% | 86.4% | 92.1% | 35.7% | 57.1% | 37.8% |
| OptiTree | **98.0%** | 84.3% | 76.7% | 94.5% | 64.3% | 50.0% | 74.8% |
| EvoSkill | 95.1% | 93.2% | 82.5% | 97.1% | 69.1% | 73.8% | 82.1% |
| OptSkills | 74.3% | 92.1% | 70.4% | 97.6% | 66.7% | 71.4% | 52.6% |
| OR-Skill(ours) | **98.0%** | **93.8%** | **90.8%** | **98.2%** | **71.4%** | **78.6%** | **91.0%** |

**Configuration.** For the agent framework, we use Claude Code. The available tools include LSP checks and Python execution. Generated programs use modeling libraries such as Pyomo or PuLP to call installed solver backends; PuLP is permitted in the case-study runs. Web search is disabled throughout evaluation to avoid answer leakage. The total number of evolution steps $T$ is set to 12, and the number of child skills per round $N$ is set to 5. The regularization weights $w_1$ and $w_2$ are set to $10^{-3}$ and $2 \times 10^{-4}$, respectively. We use Deepseek-R1 as the base model in evolutionary process, as it is a capable reasoning model available.

### 4.2 Overall Performance

Table 1 presents the results. OR-Skill achieves the highest accuracy on all seven evaluation subsets, tying with OptiTree on NL4Opt. On the challenging benchmarks, EvoSkill is the strongest baseline, with 69.1%, 73.8%, and 82.1% accuracy on ComplexOR, IndustryOR, and ComplexLP, respectively. OR-Skill improves these results to 71.4%, 78.6%, and 91.0%, corresponding to gains of 2.3, 4.8, and 8.9 percentage points. OptSkills achieves 66.7%, 71.4%, and 52.6% on the same benchmarks. These comparisons show that OR-Skill remains effective against both general skill evolution and optimization-specific skill learning, particularly on ComplexLP.

On the simpler benchmarks, the strongest baselines vary by dataset. OptiTree matches OR-Skill at 98.0% on NL4Opt, while EvoSkill reaches 93.2% on NLP4LP, close to OR-Skill's 93.8%. On EasyLP, OptSkills and EvoSkill achieve 97.6% and 97.1%, respectively, compared with 98.2% for OR-Skill. On ReSocratic, CAFA remains the strongest baseline at 86.4%, while OR-Skill reaches 90.8%. Thus, skill-based methods are competitive on several simpler benchmarks, but their strengths vary: OptSkills performs particularly well on EasyLP, whereas EvoSkill is stronger on NL4Opt and ReSocratic. OR-Skill maintains strong performance across all four datasets.

The two skill-based baselines exhibit different transfer patterns. EvoSkill is stronger than OptSkills on six of the seven evaluation subsets, including ComplexLP (82.1% versus 52.6%). Its failure-driven skill revision may help accumulate corrections across diverse modeling errors, yielding broader coverage. OptSkills achieves its highest accuracy on EasyLP (97.6%) and also performs well on NLP4LP (92.1%), consistent with archetype-based workflow distillation being effective when recurring problem structures match the learned skills. Its lower accuracy on NL4Opt, ReSocratic, and ComplexLP suggests that transfer also depends on the coverage of the skill library and the suitability of the retrieved workflows. OptSkills does refine and expand its library during learning; the observed variation therefore cannot be attributed simply to a lack of adaptation. OR-Skill's gains over both methods, especially on ComplexLP, are consistent with the benefit of consolidating modeling failures into transferable root-cause reports and coordinating corrections through cascade evolution. These

Table 2: Ablation study of OR-Skill on MAMO.

| Variant | EasyLP | ComplexLP |
|---|---|---|
| Full OR-Skill | **98.2%** | **91.0%** |
| w/o attribution analysis | 94.1% | 79.2% |
| w/o cascade evolution | 91.4% | **91.0%** |
| w/o Occam regularization | 92.2% | 87.3% |

Table 3: Performance across base models.

| Base model | LiveCodeBench | MAMO |
|---|---|---|
| Deepseek V3 | 87.5 | **94.5%** |
| Gemini 3 Pro | 86.4 | 94.2% |
| Opus 4.6 | 84.7 | 93.8% |
| Grok 4 | 83.2 | 91.5% |

are possible explanations of the benchmark-level patterns; the comparison alone does not isolate the effects of skill coverage, retrieval, and update strategy.

Among the reproduced training-based methods, DeepOR is the strongest on the three harder subsets and reaches 97.7% on NL4Opt. MURKA reports 95.5% on EasyLP in its original paper; this starred result is a reference value rather than a comparison under our cleaned-data protocol. On hard benchmarks, their performance is generally less competitive than that of the strongest skill-based agents. One possible explanation is that training-based methods rely heavily on the coverage of their training data, while complex industrial-style optimization problems are relatively sparse and diverse. As a result, models trained on limited data may struggle to generalize to realistic unseen cases. In contrast, agent-based methods with self-adaptation capabilities can better handle such scenarios.

Additionally, comparing OR-Skill with standard input-output LLM calls also shows clear gains. Since the standard baseline uses the same base model, the performance gap indicates that the improvement comes from the agent framework and the evolved skill, rather than solely from the underlying LLM.

### 4.3 Ablation Study

We evaluate three ablations on EasyLP and ComplexLP from MAMO (Table 2): removing root-cause attribution and using raw per-instance errors instead of consolidated per-level reports; replacing cascade evolution with a single pass of updates; and removing the length penalty from the selection objective. Attribution has the largest effect on ComplexLP: without it, accuracy drops from 91.0% to 79.2%. Individual error messages may emphasize surface symptoms rather than shared modeling deficiencies, giving skill updates less reliable guidance.

Removing cascade evolution yields the same reported ComplexLP accuracy (91.0%) but lowers EasyLP accuracy from 98.2% to 91.4%. Equal aggregate accuracy does not establish that the two variants solve the same instances; that requires paired instance-level outcomes. This is consistent with Section 3.7: mixing hard-case fixes with other updates in a single pass may accumulate specialized rules that complicate simpler tasks. Applying low-severity fixes last helps preserve basic modeling competence while retaining earlier corrections for difficult cases. Occam regularization serves a complementary purpose. Removing it lowers accuracy on both benchmarks, to 92.2% on EasyLP and 87.3% on ComplexLP. Without a length penalty, skills can accumulate details that distract the agent from general modeling rules. This may explain why longer skills can hurt both complex and simple problems.

### 4.4 Other LLMs as Base Models

We further evaluate OR-Skill with different base LLMs, and the results are shown in Table 3. To avoid potential judge bias, we exclude GPT from this experiment. First, OR-Skill achieves consistently strong performance across all base models, indicating that the gains brought by self-evolution are robust and are not tied to idiosyncratic features of any specific LLM API. Second, we observe a positive correlation between the coding proficiency of the base model, measured by LiveCodeBench (Jain et al., 2024), and its performance within our framework. This suggests that stronger coding ability translates into better optimization modeling performance. This is desirable, indicating that OR-Skill can directly benefit from future advances in base model capabilities.

### 4.5 Evolution Process

Figure 3 traces the accuracy trajectory on ComplexLP during the evolution process, where each point corresponds to the current top-scoring skill in the database at that iteration. At iteration 0, the initial

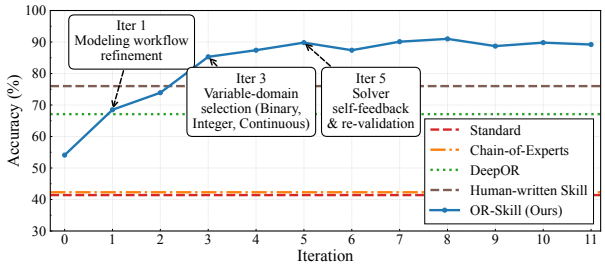



Figure 3: Available ComplexLP accuracy trajectory from iteration 0 through 11.

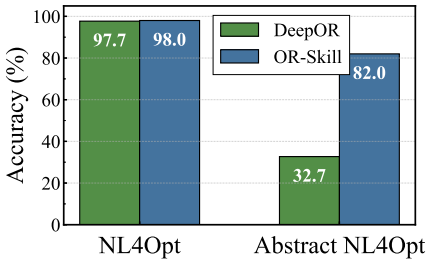



Figure 4: In-distribution vs. OOD accuracy for DeepOR and OR-Skill.

skill $s_0$ contains only a minimal meta prompt. As annotated in the figure, several key capabilities emerge through self-evolution. At iteration 1, the skill clarifies the modeling workflow by organizing it into a structured procedure. At iteration 3, it learns to distinguish among binary, integer, and continuous variable domains, which helps the agent reach an accuracy of 85.3%. At iteration 5, the skill further prompts the agent to cross-check the generated formulation through solver feedback and re-validation. These acquired capabilities help explain the gains of OR-Skill on hard instances.

We also recruited a human expert in operations research to manually curate a skill. The human-written skill performs well, achieving an accuracy of 76%, but it is surpassed by the self-evolved skill at iteration 3. We further observe that the human-written skill is much shorter than the self-evolved one (1.6k vs. 11k tokens), suggesting that manual curation can miss many corner cases and fine-grained modeling details that are discovered through self-evolution.

### 4.6 Out-of-Distribution (OOD) Generalization

We evaluate OOD generalization using an abstract NL4Opt variant with all numerical values removed from the problem descriptions, assessing modeling correctness. We compare OR-Skill with DeepOR, the strongest training-based baseline on NL4Opt. As shown in Figure 4, DeepOR drops from 97.7% to 32.7%, while OR-Skill drops from 98.0% to 82.0%, a decrease of 16.0 percentage points. OR-Skill is more robust to numerical abstraction in this comparison. These results do not establish a general advantage of all agent-based methods over training-based models or identify the cause of the difference.

### 4.7 Case Study

The case studies in Appendix D illustrate verification and revision behavior. In one logistics problem, the agent enumerates all selected mode pairs to verify optimality under the reference interpretation (Appendix D.2). In another, an initially pollution-optimal solution uses 10 truck trips and 1,000 EV trips to transport 61,000 units for a demand of 1,800. A secondary trip penalty reduces EV use to 14 trips while retaining the minimum pollution of 1,000 (Appendix D.3). The change improves the choice among optimal solutions rather than correcting a zero-pollution answer.

## 5 Conclusion

We presented OR-Skill, a self-evolution framework for optimization modeling agents that evolves skill artifacts instead of relying on manual expert redesign. The method combines git-versioned skill repositories with commit-based updates, four-level quality grading, per-level root-cause attribution, a reverse-order cascade of edits, and Occam-regularized selection over a skill database. Experiments show higher accuracy than prior agentic, training-based, and direct-calling baselines, with consistent gains across stronger base models. Overall, our findings suggest that self-evolution is an effective way to adapt general-purpose agents to optimization modeling.

## AI USE STATEMENT

Generative AI models are used to implement experiment programs, including the optimization algorithm and LLM judger, as described in Section 3 and Appendix A.2. OpenAI Codex assisted with LaTeX formatting, layout, minor wording edits, and consistency checks.

## REPRODUCIBILITY STATEMENT

Section 3 and Algorithm 1 describe the skill-evolution procedure. Section 4.1 reports the evaluation setup. Appendix A provides execution details, scoring conventions, and prompt templates; Appendix C reports additional ablations and evaluation checks; and Appendix E describes computational resources.

## ETHICS STATEMENT

OR-Skill can produce incorrect assumptions, objectives, or constraints even when its generated program executes successfully. In consequential applications, domain experts should review the formulation and its outputs before decisions are made. Appendix G discusses these risks.

## REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (MI)LP solvers and large language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=YT1dtdLvSN.

Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. EvoSkill: Automated skill discovery for multi-agent systems. *CoRR*, abs/2603.02766, 2026. doi: 10.48550/ARXIV.2603.02766. URL https://doi.org/10.48550/arXiv.2603.02766.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*, pp. 1864–1886. PMLR, 2025. URL https://proceedings.mlr.press/v267/astorga25a.html.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pp. 17682–17690. AAAI Press, 2024. doi: 10.1609/AAAI.V38I16.29720. URL https://doi.org/10.1609/aaai.v38i16.29720.

Gabriel R. Bitran, Elizabeth A. Haas, and Hirofumi Matsuo. Production planning of style goods with high setup costs and forecast revisions. *Oper. Res.*, 34(2):226–236, 1986. doi: 10.1287/OPRE.34.2.226. URL https://doi.org/10.1287/opre.34.2.226.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling, 2025. URL https://arxiv.org/abs/2505.11792.

Zezhen Ding, Zhen Tan, Jiheng Zhang, and Tianlong Chen. OR-R1: automating modeling and solving of operations research optimization problem via test-time reinforcement learning. In Sven Koenig, Chad Jenkins, and Matthew E. Taylor (eds.), *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*,

pp. 228–236. AAAI Press, 2026. doi: 10.1609/AAAI.V40I1.36983. URL https://doi.org/10.1609/aaai.v40i1.36983.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*. PMLR, 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

haoxuan deng, Bohao Zheng, Yirui Jiang, and Trung Hieu Tran. CAFA: Coding as auto-formulation can boost large language models in solving linear programming problem. In *The 4th Workshop on Mathematical Reasoning and AI at NeurIPS'24*, 2024. URL https://openreview.net/forum?id=xC2xtBLmri.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *CoRR*, abs/2405.17743, 2024a. doi: 10.48550/ARXIV.2405.17743. URL https://doi.org/10.48550/arXiv.2405.17743.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Mamo: a mathematical modeling benchmark with solvers. *CoRR*, abs/2405.13144, 2024b. doi: 10.48550/ARXIV.2405.13144. URL https://doi.org/10.48550/arXiv.2405.13144.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

Haoyang Liu, Jie Wang, Yuyang Cai, Xiongwei Han, Yufei Kuang, and Jianye HAO. Optitree: Hierarchical thoughts generation with tree search for llm optimization modeling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. Optmath: A scalable bidirectional data synthesis framework for optimization modeling. *CoRR*, abs/2502.11102, 2025. doi: 10.48550/ARXIV.2502.11102. URL https://doi.org/10.48550/arXiv.2502.11102.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery. *CoRR*, abs/2506.13131, 2025. doi: 10.48550/ARXIV.2506.13131. URL https://doi.org/10.48550/arXiv.2506.13131.

OpenAI. Openai GPT-5 system card. *CoRR*, abs/2601.03267, 2026. doi: 10.48550/ARXIV.2601.03267. URL https://doi.org/10.48550/arXiv.2601.03267.

Xiangyu Peng, Congying Xia, Xinyi Yang, Caiming Xiong, Chien-Sheng Wu, and Chen Xing. Regenesis: LLMs can grow into reasoning generalists via self-improvement. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=YUYJsHOf3c.

William P. Pierskalla and David J. Brailer. Chapter 13 applications of operations research in health care delivery. In Stephen M. Pollock, Michael H. Rothkopf, and Arnold Barnett (eds.), *Operations research and the public sector*, Handbooks in operations research and management science, pp. 469–505. North-Holland, 1994. doi: 10.1016/S0927-0507(05)80094-5. URL https://doi.org/10.1016/s0927-0507(05)80094-5.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *CoRR*, abs/2505.20286, 2025. doi: 10.48550/ARXIV.2505.20286. URL https://doi.org/10.48550/arXiv.2505.20286.

Rindranirina Ramamonjison, Timothy T. L. Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. *CoRR*, abs/2303.08233, 2023. doi: 10.48550/ARXIV.2303.08233. URL https://doi.org/10.48550/arXiv.2303.08233.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *CoRR*, abs/2302.04761, 2023. doi: 10.48550/ARXIV.2302.04761. URL https://doi.org/10.48550/arXiv.2302.04761.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mirac Suzgun, Mert Yüksekgönül, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory. *CoRR*, abs/2504.07952, 2025. doi: 10.48550/ARXIV.2504.07952. URL https://doi.org/10.48550/arXiv.2504.07952.

Constantine Toregas, Ralph Swain, Charles S. ReVelle, and Lawrence Bergman. The location of emergency service facilities. *Oper. Res.*, 19(6):1363–1373, 1971. doi: 10.1287/OPRE.19.6.1363. URL https://doi.org/10.1287/opre.19.6.1363.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *CoRR*, abs/2305.16291, 2023. doi: 10.48550/ARXIV.2305.16291. URL https://doi.org/10.48550/arXiv.2305.16291.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. doi: 10.48550/ARXIV.2201.11903. URL https://doi.org/10.48550/arXiv.2201.11903.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Yang Wu, Yifan Zhang, Yurong Wu, Yuran Wang, Junkai Zhang, and Jian Cheng. Training LLMs for optimization modeling via iterative data synthesis and structured validation. In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 5-9, 2025*, pp. 12880–12896. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-emnlp.691. URL https://aclanthology.org/2025.findings-emnlp.691/.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-experts: When LLMs meet complex operations research problems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Ziyang Xiao, Jingrong Xie, Lilin Xu, Shisi Guan, Jingyan Zhu, Xiongwei Han, Xiaojin Fu, WingYin Yu, Han Wu, Wei Shi, Qingcan Kang, Jiahui Duan, Tao Zhong, Mingxuan Yuan, Jia Zeng, Yuan Wang, Gang Chen, and Dongxiang Zhang. A survey of optimization modeling meets llms: Progress and future directions. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025*, pp. 10742–10750. ijcai.org, 2025. doi: 10.24963/IJCAI.2025/1192. URL https://doi.org/10.24963/ijcai.2025/1192.

Ziyang Xiao, Yuan Jessica Wang, Xiongwei Han, Shisi Guan, Jingyan Zhu, Jingrong Xie, Lilin Xu, Han Wu, Wing Yin Yu, Zehua Liu, Xiaojin Fu, Gang Chen, and Dongxiang Zhang. Deepor: A deep reasoning foundation model for optimization modeling. In Sven Koenig, Chad Jenkins, and Matthew E. Taylor (eds.), *Fortieth AAAI Conference on Artificial Intelligence, Thirty-Eighth Conference on Innovative Applications of Artificial Intelligence, Sixteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2026, Singapore, January 20-27, 2026*, pp. 34052–34060. AAAI Press, 2026. doi: 10.1609/AAAI.V40I40.40699. URL https://doi.org/10.1609/aaai.v40i40.40699.

Wantong Xie, Yi-Xiang Hu, Jieyang Xu, Feng Wu, and Xiangyang Li. MURKA: Multi-reward reinforcement learning with knowledge alignment for optimization tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=f4pvPNf9ox.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *CoRR*, abs/2602.12430, 2026. doi: 10.48550/ARXIV.2602.12430. URL https://doi.org/10.48550/arXiv.2602.12430.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *CoRR*, abs/2309.03409, 2023. doi: 10.48550/ARXIV.2309.03409. URL https://doi.org/10.48550/arXiv.2309.03409.

Haochen Yang, Ke Zhao, Mengyuan Ma, Xingyu Lu, Xiangfeng Wang, and Hong Qian. OptSkills: Learning generalizable optimization skills from problem archetypes via cluster-based distillation. *CoRR*, abs/2605.29829, 2026. doi: 10.48550/ARXIV.2605.29829. URL https://doi.org/10.48550/arXiv.2605.29829.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Wei Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve LLMs for optimization modeling. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=fsDZwS49uY.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *CoRR*, abs/2210.03629, 2022. doi: 10.48550/ARXIV.2210.03629. URL https://doi.org/10.48550/arXiv.2210.03629.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *CoRR*, abs/2305.10601, 2023. doi: 10.48550/ARXIV.2305.10601. URL https://doi.org/10.48550/arXiv.2305.10601.

Mert Yüksekgönül, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. *CoRR*, abs/2406.07496, 2024. doi: 10.48550/ARXIV.2406.07496. URL https://doi.org/10.48550/arXiv.2406.07496.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. *CoRR*, abs/2510.04618, 2025. doi: 10.48550/ARXIV.2510.04618. URL https://doi.org/10.48550/arXiv.2510.04618.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, Huajun Chen, and Yuchen Eleanor Jiang. Symbolic learning enables self-evolving agents. *CoRR*, abs/2406.18532, 2024. doi: 10.48550/ARXIV.2406.18532. URL https://doi.org/10.48550/arXiv.2406.18532.

# A Implementation Details

## A.1 Agent Execution Protocol

Each modeling task is executed in a shared sandbox directory that is cleaned between runs, retaining only the skill repository. The natural-language problem description is provided to the agent as the initial prompt, along with the currently selected skill repository. The agent is instructed to save its final executable optimization model as `model.py`. The harness archives that submitted program and its execution logs; auxiliary debugging files are not automatically treated as final solutions. In Case Study 4, the agent transfers its simplified formulation to `model.py` and reruns it before submission.

During both evolution and evaluation, web search is disabled to avoid answer leakage. The available tools include file editing, command-line execution, Python execution, and LSP checks. Pyomo and PuLP are modeling interfaces rather than solvers; generated programs invoke a solver backend through these libraries. The archived PuLP cases use CBC, while the Pyomo examples use GLPK. The answer directory is renamed to reduce accidental access. This naming convention alone does not enforce a filesystem access boundary. Each generated program is required to print the final objective value and, when available, the relevant decision-variable assignment. Execution traces (standard output, standard error, solver status, and runtime information) are logged and passed to the evaluator.

The evaluation scripts use a default 300-second (5-minute) limit for re-executing the final Python program. Agent generation and debugging form a separate stage whose runtime includes model calls and tool use; the program-execution limit should not be interpreted as an end-to-end agent timeout. Solver runtime depends on the generated formulation and is not guaranteed to be a few seconds.

## A.2 LLM Calls in the Evolution Loop

Table 4 summarizes the major LLM calls used in one round of self-evolution. The loop has four types of LLM calls: task execution by the coding agent, model judgment, root-cause attribution, and skill editing. The dominant cost comes from executing and scoring candidate skills on Hard100.

Table 4: Scheduled operations in one self-evolution round. Here $|\mathcal{D}| = 100$ is the size of Hard100 and $N = 5$ is the number of child skills. An agent execution is a complete task run and can contain multiple LLM API calls.

| Module | Input | Output | Model | Operations per round |
|---|---|---|---|---|
| Agent execution | Problem $P_i$, skill $s$ | Generated model $\widehat{M}_i$ and execution trace | DeepSeek V3.2 | $(1+N)\|\mathcal{D}\|$ |
| Judgment | $(P_i, \widehat{M}_i, M_i)$ and execution trace | Level $L_0$–$L_3$ and explanation | GPT-5 | $(1+N)\|\mathcal{D}\|$ |
| Root-cause attribution | Error cases grouped by level | Level-wise diagnostic reports | Deepseek-R1 | 3 |
| Cascade evolution | Skill repository and diagnostic report | Git patch / commit | Deepseek-R1 | $3N$ |
| Scoring | Accuracy and skill length | Occam-regularized score | - | $N$ |

## A.3 Scoring Details

We detail the computation of the Occam-regularized score defined in Equation 3.

**Accuracy.** $\mathrm{Acc}(s)$ is the fraction of problems in Hard100 for which the generated program produces a correct objective value within a relative tolerance of 0.01.

**Token counting.** $N_{\mathrm{meta}}(s)$ and $N_{\mathrm{ref}}(s)$ are measured in GPT-4 tokens using the `tiktoken` library with the `cl100k_base` encoding. We count `SKILL.md` and all files under `references/` separately so that the two penalty weights can balance meta-level instructions against domain-specific

content. Files under `scripts/` are excluded because executable utilities do not consume the agent's context window during modeling.

**Weight selection.** We set $w_1 = 10^{-3}$ and $w_2 = 2 \times 10^{-4}$. The weights differ because every token in `SKILL.md` is always loaded, whereas reference tokens are only conditionally retrieved. With accuracy expressed as a fraction in $[0, 1]$, adding 100 tokens to `SKILL.md` increases the penalty by 0.1. On Hard100, this equals the accuracy contribution of 10 correctly solved problems (10 percentage points); one problem contributes 0.01.

### A.4 Prompt Templates

Core templates follow. Each prompt is composed dynamically at runtime. The system message provides role instructions and output schema constraints, while the user message is populated with instance-specific content such as problem text, generated code, and execution traces. All LLM calls use structured JSON output mode where available; when the provider does not support it natively, we append explicit schema examples and strip non-JSON preambles during post-processing.

**Modeling agent prompt.** The modeling agent receives a system prompt derived from the skill's `SKILL.md`, which defines the workflow and domain heuristics. The user message contains the problem text and a target file path:

**Modeling Agent Prompt**

You are an operations research modeling expert. Solve the following OR problem.
Requirements:
- Fully understand the problem; construct the correct mathematical model, including decision variables, objective, and constraints.
- Produce a self-contained executable `model.py`. - The file must include data definitions, model construction, a solver call, and result output.
- Define exactly one entry-point function `solve()` that returns the optimal objective value as a number.
- After generation, execute the code, inspect solver status and solution feasibility, and revise if the output is invalid.

The agent operates inside a sandbox with the skill available through its context directory. It can read files under `references/` and invoke scripts under `scripts/`. The prompt requests a `solve()` entry point. Archived case-study scripts that print their result directly do not always follow this entry-point convention.

**Judgment prompt.** The judge receives the problem description, the generated program, its execution trace (stdout and stderr), and the expert reference model. The system prompt specifies the four-level rubric and a mandatory JSON output schema.

**Judgment Prompt**

Given a problem description, the generated program, its execution trace, and the expert reference model, classify the solution into one of four levels:
$L_0$ Correct — solver output matches the optimal value within relative tolerance 0.01;
$L_1$ Syntax Error — code fails to execute or solver returns infeasible/error;
$L_2$ Improvable — model runs and is partially correct but contains debatable choices (e.g., overly loose constraints, ambiguous assumptions);
$L_3$ Logical Error — code executes successfully but the optimal answer is wrong.
Procedure: First explain the modeling behavior. Then compare the generated formulation against the reference model, focusing on structural differences, such as missing constraints, wrong objective, and incorrect domains. Finally output JSON:
`{"level": "L0"|"L1"|"L2"|"L3", "reason": "..."}`.

The judge is called with temperature 0 to maximize determinism. We enforce `response_format: json_object` when the provider supports it.

**Root-cause attribution prompt.** Attribution is performed once per error level. The system prompt enforces cluster-level reasoning and prohibits instance-level summarization.

**Root-Cause Attribution Prompt**

You are given $k$ error cases that all belong to the same error level. Do not summarize them one by one. Instead, cluster them by underlying root cause, such as "fails to model piecewise cost" or "confuses min-cost flow with shortest path". For each cluster, provide:
(1) the missing modeling capability;
(2) a concise explanation of why it causes failure;
(3) a general rule that can help solve unseen problems exhibiting the same pattern.
Avoid hard-coding numerical answers, instance names, or solution values. Output JSON:

```
{
 "clusters":[
 {
 "cause": ...,
 "capability_gap": ...,
 "rule": ...,
 "affected_problems": [...]
 }
 ]
}.
```

The attribution LLM is called with temperature 0.5 and a context budget of up to 32k input tokens per level. When the error set for a level exceeds this budget, we prioritize problems whose generated code is shortest, as they tend to exhibit the clearest failure signal, and truncate execution logs to the last 4,000 characters.

**Cascade evolution prompt.** The evolution LLM receives the full current skill repository, serialized as a JSON object mirroring the directory hierarchy, together with the level-wise diagnostic reports. The system prompt reads:

**Cascade Evolution Prompt**

You are given the current skill repository and a consolidated diagnostic report for one error level. Improve the skill so that the identified failure patterns are addressed. Guidelines:
(1) Prefer concise, general, and reusable rules over instance-specific patches.
(2) If the fix belongs to a specific problem family, add or update the corresponding file under `references/` rather than bloating `SKILL.md`.
(3) Do not insert numerical answers, dataset-specific identifiers, or memorized solutions.
(4) Return a valid unified diff that can be applied via `git apply`. Each hunk should have a clear commit message prefix.

The evolution LLM is called with temperature 0.6 and a generous output budget of up to 12,000 tokens to accommodate multi-file diffs.

### A.5 Example submitted Git diff.

To make the skill update format concrete, we show an example unified diff produced by the evolution LLM. The patch edits the top-level `SKILL.md` and adds a domain-specific reference file under `references/`. This example illustrates the expected format: an atomic, auditable change that can be directly applied by `git apply`.

**Example Submitted Git Diff**

```
diff --git a/SKILL.md b/SKILL.md
index 8f3a21c..b7d9c42 100644
--- a/SKILL.md
+++ b/SKILL.md
@@ -10,6 +10,9 @@ When solving an optimization modeling problem:
 2. Define decision variables before writing constraints.
 3. Check the objective direction before solving.
 4. Verify whether each variable has the correct domain.

+5. For multi-period inventory problems, explicitly check whether unmet
+   demand is lost, penalized once, or carried forward as backlog.
+
 ## Self-check

 Before returning the final answer, execute the model and inspect the solver status.
```

# B Analysis of the Evolved Skill

## B.1 Final Skill Structure

The final evolved skill is a self-contained, three-layer directory. A top-level instruction file provides the general workflow, domain-specific reference guides supply specialized modeling knowledge, and an executable validation script handles solver invocation. Figure 5 shows the full file layout.

```
skill/
|-- SKILL.md                      # Core workflow, problem analysis
|                                 # framework, failure-mode guardrails,
|                                 # and code generation principles
|-- references/
|   |-- prompt_template.md        # Baseline LLM prompt template with
|   |                             # pre-submission checklist
|   |-- network_flow_problems.md
|   |                             # Max-flow, shortest-path, min-cost
|   |                             # flow; net-flow formulation pitfalls
|   |-- workforce_inventory_planning.md
|   |                             # Multi-period balance equations,
|   |                             # initial/terminal conditions
|   |-- logical_modeling_patterns.md
|   |                             # Exact-one vs at-most-one logic,
|   |                             # big-M tightening, activation
|   `-- data_mapping_and_diet_problems.md
|                                 # Data extraction from tables, unit
|                                 # conversion, bound-direction checks
`-- scripts/
    `-- run_solver_module.py      # Loads generated code, validates
                                  # solve() signature, executes solver,
                                  # checks numeric finiteness of result
```

Figure 5: Directory layout of the evolved OR modeling skill. Each reference file is consulted conditionally based on the detected problem type.

The `SKILL.md` serves as the primary instruction, defining a structured modeling workflow, enumerating high-risk failure modes (truncated code, wrong network objectives, loose big-M values, data mapping errors), and specifying the output contract. Detailed domain knowledge is placed in reference files to avoid overloading the main instruction. This design follows progressive disclosure: the agent first receives general guidance from `SKILL.md` and then accesses specialized references only when the detected problem type requires them (e.g., consulting `network_flow_problems.md` only for flow/path problems).

### B.2 Categories of Acquired Modeling Knowledge

The evolved skill acquires several categories of transferable modeling knowledge through repeated diagnosis and skill editing during self-evolution. Table 5 summarizes them, each traceable to specific sections of the final skill files.

Table 5: Categories of knowledge acquired by the evolved skill.

| Category | Failure prevented | Source in skill |
|---|---|---|
| Variable-domain selection | Relaxed or over-discretized formulations | `SKILL.md` |
| Network flow modeling | Invalid max-flow; wrong source/sink balance | `network_flow_problems.md` |
| Workforce & inventory planning | Missing boundary conditions; wrong recursion | `workforce_inventory_planning.md` |
| Logical & selection modeling | Wrong exclusivity semantics; oversized big-$M$ | `logical_modeling_patterns.md` |
| Data mapping & diet problems | Column-shift errors; swapped $\geq$/$\leq$ bounds | `data_mapping_and_diet_problems.md` |
| Solver status validation | Reading objective from failed/infeasible solve | `SKILL.md` |
| Code completeness | Truncated code; missing `solve()` entrypoint | `SKILL.md` |

These capabilities are not inserted manually. They emerge through the iterative self-evolution loop. Early iterations primarily improve the agent's general workflow and code-completeness checks, while later iterations introduce domain-specific guardrails, such as the net-flow $v$-variable pattern, tight big-M derivation rules, and dictionary-keyed data extraction, that target recurring error classes identified by the attribution mechanism.

### B.3 Skill Evolution Trajectory

To better understand how the skill improves, we inspect the Git history of the best-performing skill. Table 6 lists representative commits and the modeling capability each introduces.

Table 6: Representative skill updates during self-evolution.

| Iteration | Main update |
|---|---|
| 1 | Structured modeling workflow |
| 2 | Network flow and logical modeling reference files |
| 3 | Rules for binary vs. integer variables; workforce planning reference |
| 4 | Data mapping and diet problems reference |
| 5 | Self-feedback and cross-check procedure |
| 6 | Objective-sense preservation rules (cost-only $\Rightarrow$ minimize) |
| 7 | Tight big-$M$ derivation rules |
| 8 | Initial/terminal boundary condition enforcement for multi-period models |
| 9 | Pyomo code examples for net-flow $v$-variable pattern |
| 10 | Solver-status and numeric-finiteness validation in runner |
| 11 | Constraint-direction semantic checks |
| 12 | Dictionary-keyed data extraction; completeness checklist |

This trajectory suggests that self-evolution first improves general modeling discipline (iterations 1–5 correspond to workflow formalization, variable-domain rules, and cross-checking) and then gradually accumulates domain-specific modeling knowledge (iterations 7–12 cover tight big-$M$ derivation, net-flow formulation patterns, and dictionary-keyed data extraction). The resulting skill is therefore not merely a longer prompt but a structured repository of reusable modeling behavior.

### B.4 Token Distribution of the Skill

We analyze the token distribution of the final evolved skill (measured with the GPT-4 tokenizer). Table 7 reports the token counts in each component.

Table 7: Token distribution of the evolved skill.

| Component | Number of files | Tokens | Percentage |
|---|---|---|---|
| `SKILL.md` | 1 | 3,817 | 32.3% |
| `references/` | 5 | 7,427 | 62.8% |
| `scripts/` | 1 | 584 | 4.9% |
| Total | 7 | 11,828 | 100% |

The distribution reveals a clear modular pattern. The `SKILL.md` accounts for only about one-third of the total budget, serving as a compact dispatch layer that defines the general workflow and failure-mode guardrails. The majority of tokens (62.8%) reside in domain-specific reference files, with `network_flow_problems.md` being the largest (2,815 tokens) due to its detailed Pyomo examples and multiple formulation variants. The `scripts/` component is minimal (4.9%), containing only the solver-execution harness.

This distribution shows that the evolved skill does not degenerate into a monolithic prompt. Instead, it follows a structured hierarchy where general rules reside in the main file, specialized knowledge lives in conditionally-loaded references, and executable validation is handled by scripts. For each problem, the agent uses progressive disclosure to load `SKILL.md` plus at most one or two relevant reference files, keeping the effective context well within typical LLM window constraints.

## C Additional Experiments

### C.1 Component Ablation of the Skill

To understand which component contributes most to performance, we conduct a component-level ablation. Starting from the final skill, we evaluate variants with references removed, scripts removed, or both removed. The last variant is labeled "Only `SKILL.md` and retains only the workflow instruction file. The experiment distinguishes between general workflow guidance, domain references, and executable validation. When removing references or scripts, we also remove their mentions in `SKILL.md` to avoid dangling references. The bare agent receives none of the skill components.

Table 8: Component ablation of the evolved skill.

| Variant | EasyLP | ComplexLP |
|---|---|---|
| Full skill | 98.2% | 91.0% |
| Only `SKILL.md` | 91.9% | 62.2% |
| w/o references | 93.8% | 76.6% |
| w/o scripts | 91.4% | 80.2% |
| Bare agent | 92.1% | 56.8% |

The results are shown in Table 8. First, the full skill yields consistent gains on both easy and hard benchmarks (+6.1 percentage points on EasyLP and +34.2 on ComplexLP vs. the bare agent), with the benefit most pronounced on challenging instances.

Among single-component removals, removing scripts causes the larger decline on EasyLP (91.4%), while removing references causes the larger decline on ComplexLP (76.6%). Removing both components leaves only `SKILL.md` and reduces ComplexLP accuracy further to 62.2%. These results are consistent with different roles for executable checks and domain-specific references; they do not identify self-verification as the sole bottleneck or establish that references themselves introduce noise.

### C.2 Error Distribution Before and After Evolution

We analyze how the distribution of error types changes as the skill evolves. Figure 6 visualizes the proportion of each error level on Hard100 at three evolution stages.

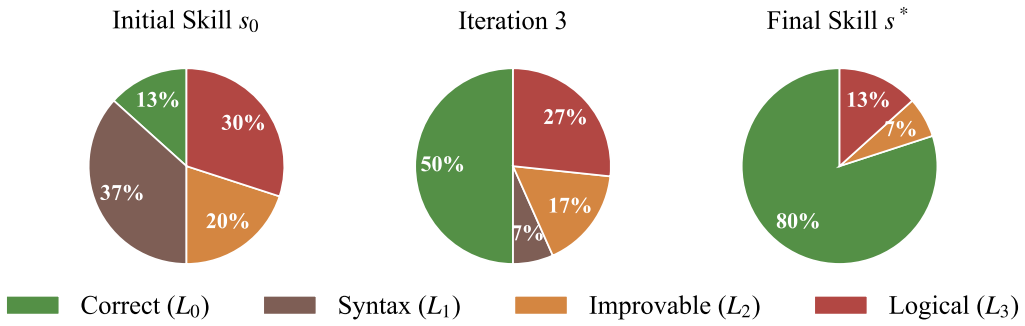



Figure 6: Error distribution across evolution stages (original diagnostic snapshot; counts pending verification for Hard100). The skill evolution progressively eliminates syntax errors first, then reduces logical and improvable errors.

The evolution exhibits a clear stage-wise pattern. Early iterations primarily suppress syntax errors ($L_1$), reducing them from 37% initially to near-zero by iteration 3 as the skill acquires basic code-generation discipline. In later iterations, the focus shifts to modeling correctness, reducing logical errors ($L_3$) from 30% to 13% and improvable cases ($L_2$) from 20% to 7%. This ordering aligns with the curriculum structure of our evolution: since syntactic validity is a prerequisite for meaningful semantic feedback, the judge naturally prioritizes $L_1$ fixes before addressing deeper $L_3$ issues.

### C.3 Judge Robustness

Because both evolution and final evaluation rely on LLM-based judgment, we conduct a robustness check. We compare the default judge (GPT-5) with two alternative evaluation methods: (i) a stronger LLM judge (GPT-5-pro), and (ii) Extract Answer Match, which uses regex-based numerical extraction to parse the solver output and compares it against the ground-truth value within relative tolerance 0.01. We additionally recruit two operations research experts to independently annotate a stratified subset of 50 evaluation instances (drawn proportionally from all seven evaluation subsets) as the human gold standard.

Table 9: Judge robustness analysis. Human agreement is measured on the 50-instance expert-audited subset (Cohen's $\kappa$). Accuracy is reported on ComplexLP (full benchmark).

| Judge | Human Agreement ($\kappa$) | OR-Skill Acc. | OptiTree Acc. |
|---|---|---|---|
| Human expert consensus | — | 91.9% | 73.9% |
| Default judge (GPT-5) | 0.92 | 91.0% | 74.8% |
| GPT-5-pro | 0.94 | 91.9% | 75.7% |
| Extract Answer Match | 0.61 | 82.0% | 68.5% |

Several conclusions emerge from Table 9. First, the default judge (GPT-5) achieves high agreement with human experts ($\kappa = 0.92$), and the measured accuracy of both OR-Skill and OptiTree is nearly identical to the human consensus (within 1 percentage point). This confirms that the LLM-based evaluation faithfully reflects true solution quality. Second, upgrading to GPT-5-pro yields only marginal improvement in human agreement ($\kappa = 0.94$) and produces accuracy estimates within 1% of the default judge. The relative ranking between OR-Skill and OptiTree is fully preserved, demonstrating that the judgment task does not require the strongest available model and that GPT-5 already saturates the evaluation quality for this domain.

In contrast, Extract Answer Match shows substantially lower agreement with human experts ($\kappa = 0.61$) and systematically underestimates accuracy for both methods. On ComplexLP, it reduces

OR-Skill's measured accuracy from $91.0\%$ to $82.0\%$ and OptiTree's from $74.8\%$ to $68.5\%$. These discrepancies arise from the inherent limitations of pattern-based extraction, which motivate our LLM-judge design (Section 3.5). Heterogeneous output formats across benchmarks make reliable parsing difficult; for example, some programs print both decision-variable assignments and optimal values, causing the extractor to retrieve an intermediate quantity. Additionally, equivalent formulations that yield the same optimal value but differ in numerical presentation (e.g., different scaling or units) are incorrectly marked as failures, and problems admitting multiple optimal solutions cannot be verified by a single ground-truth number. These results justify our design choice of using an LLM judge and confirm that the reported performance gains are robust to the specific evaluation method.

### C.4 Data Leakage Check

To reduce the risk of data leakage, Hard100 is used only for skill evolution and is never included in the final benchmark evaluation. We additionally perform a systematic similarity check between Hard100 and all evaluation instances. We encode each problem description using the text-embedding-3-large model[2] and compute the pairwise cosine similarity between every evaluation instance and every problem in Hard100. For each evaluation benchmark, we report the maximum similarity score observed across all pairings, as well as the result of manual inspection on the top-5 most similar pairs.

Table 10: Data leakage check between Hard100 and evaluation benchmarks. Similarity is measured as the maximum cosine similarity (text-embedding-3-large) between any evaluation instance and any problem in Hard100. A near-duplicate threshold of $0.70$ is used following standard practice.

| Benchmark | Max similarity to Hard100 | Manual inspection result |
|---|---|---|
| NL4Opt | 0.23 | No near-duplicate found |
| NLP4LP | 0.28 | No near-duplicate found |
| ReSocratic | 0.19 | No near-duplicate found |
| EasyLP | 0.31 | No near-duplicate found |
| ComplexOR | 0.34 | No near-duplicate found |
| IndustryOR | 0.37 | No near-duplicate found |
| ComplexLP | 0.33 | No near-duplicate found |

All maximum similarity scores fall well below the standard near-duplicate threshold of $0.70$, and manual inspection confirms that no evaluation instance shares the same problem structure or numerical parameters with any problem in Hard100. The highest observed score ($0.37$, IndustryOR) corresponds to a pair that merely shares the same broad problem category (multi-period production planning) but differs entirely in constraints, cost structure, and scale. The moderate similarity values for the harder benchmarks (ComplexOR, IndustryOR, ComplexLP) are expected, since both Hard100 and these benchmarks draw from the same general domain of operations research. However, topical overlap at the domain level does not constitute data leakage.

[2] https://developers.openai.com/api/docs/models/text-embedding-3-large

# D CASE STUDIES

We present five case studies from archived evolution traces. The submitted programs are executable Python scripts with printed results; some do not expose the `solve()` entry point requested by the prompt template. Intermediate debugging files are distinguished from the submitted `model.py`.

## D.1 CASE STUDY 1: SELF-CHECKING VIA OUTPUT CONTRADICTION DETECTION

**INPUT**

A 9-node directed network represents a water distribution system connecting reservoirs. Each arc has a known capacity (in millions of gallons per day), and arcs exist in both directions between most node pairs, including arcs into the source (Reservoir 0) and out of the sink (Reservoir 8). The task is to find the maximum flow from source to sink.

**AGENT REASONING**

**Step 1.** The agent formulates a maximum-flow LP using PuLP with decision variables $x_{ij}$ for each arc, capacity upper bounds, and flow conservation at intermediate nodes. Objective: $\max \sum_j x_{0,j}$ (total outflow from source). Solver returns 72.
**Step 2.** The agent writes a separate verification script that computes net flow at each node by summing solved arc values. Output: source net outflow $= 54$, sink net inflow $= 54$.
**Step 3.** The agent observes the discrepancy (solver objective $= 72$ but verification reports net flow $= 54$). It prints: "The 18-unit gap equals total inflow returning to source via arcs $(3 \to 0)$, $(4 \to 0)$, $(7 \to 0)$. The objective computes gross outflow, not net outflow."
**Step 4.** The agent rewrites the objective to $\max\left(\sum_j x_{0,j} - \sum_i x_{i,0}\right)$ and adds a constraint equating source net outflow to sink net inflow. Re-solves. Both solver and verification now report 72.

**OUTCOME:** **Correct** (72.00, matching reference)

**Analysis.** The verification program was constructed by the agent itself as an independent cross-check. For each node $k$, it sums all solved arc values $x_{ik}$ and $x_{kj}$ and reports the difference. This is logically sound and would correctly detect any flow-conservation violation. However, the first model's objective was mathematically misaligned with the verification's definition of "flow." The solver was technically correct under its own objective ($\max$ gross outflow $= 72$), but the verification measured net flow ($= 54$). By detecting this $72 \neq 54$ contradiction, the agent identified a conceptual modeling error rather than a syntax bug. After correction, the model's objective, its constraints, and its verification all become mutually consistent. This demonstrates a "trust-but-verify" pattern where the agent autonomously constructs a verification oracle, detects inconsistency, diagnoses the root cause (gross vs. net flow in the presence of back-arcs), and revises the formulation.

## D.2 CASE STUDY 2: SEMANTIC DISAMBIGUATION VIA EXHAUSTIVE ENUMERATION

**INPUT**

A logistics company selects two of three transportation modes (motorcycle, small truck, large truck) and determines trip counts to minimize total pollution. The reference formulation treats selection as availability: a selected mode may receive zero trips. Motorcycles produce 40 units of pollution per trip and carry 10 units; small trucks produce 70 and carry 20; large trucks produce 100 and carry 50. Constraints include: motorcycle trips $\leq 8$, total cargo $\geq 300$ units, total trips $\leq 20$.

AGENT REASONING

**Step 1.** The agent formulates a mixed-integer program: binary variables $y_1, y_2, y_3$ (mode selection), integer variables $x_1, x_2, x_3$ (trip counts), cardinality constraint $y_1 + y_2 + y_3 = 2$, big-M linking $x_i \leq My_i$, objective $\min\ 40x_1 + 70x_2 + 100x_3$.
**Step 2.** Solver returns 600: selected modes = {small truck, large truck}, with $x_2 = 0$, $x_3 = 6$. Small truck is selected but assigned zero trips.
**Step 3.** The agent prints: "Small truck has 0 trips. Let me verify by enumerating all pairs." It constructs a separate routine that fixes each of the three mode pairs and solves independently.
**Step 4.** Enumeration results: {motorcycle, small truck} = 1050; {motorcycle, large truck} = 600; {small truck, large truck} = 600. Two pairs tie at 600. The agent confirms 600 as the global optimum.

**OUTCOME:** **Correct** (600, matching reference)

**Analysis.** In iteration 1, the agent initially obtained 600 but then required each selected mode to make at least one trip by adding $x_i \geq y_i$, raising pollution to 640. Under the reference interpretation, two modes are selected but only one needs to be used, so this additional constraint excludes valid solutions. The natural-language phrase alone can support different interpretations; the comparison here is with the benchmark's reference formulation.

In iteration 3, the agent retains the reference interpretation and enumerates all three available mode pairs. Enumeration verifies the optimum under that interpretation; it does not resolve the linguistic ambiguity by itself.

### D.3 CASE STUDY 3: COMMON-SENSE DETECTION OF DEGENERATE SOLUTIONS

INPUT

A company must transport at least 1,800 units of product. Four modes are available: trucks (100 units/trip, 100 pollution), vans (80 units, 50 pollution), motorcycles (40 units, 10 pollution), and electric vehicles (60 units, 0 pollution). Vans and EVs are mutually exclusive. Total pollution must not exceed 2,000 units, and trucks must be used at least 10 times. The objective is to minimize total pollution.

AGENT REASONING

**Step 1.** The agent builds a PuLP model with integer trip variables for each mode, binary mutual-exclusion for vans vs. EVs, constraint total cargo $\geq$ 1800, trucks $\geq$ 10, pollution $\leq$ 2000. Objective: min total pollution.
**Step 2.** Solver returns pollution = 1,000 with 10 truck trips and 1,000 EV trips, transporting 61,000 units in total.
**Step 3.** The agent observes that EVs alone transport 60,000 units. EV trips carry no pollution cost, so the primary objective does not penalize their excess use. The implemented big-$M$ constraint bounds EV trips by 1,000; the model is not unbounded.
**Step 4.** The agent adds $0.01 \times$ (total trips) to the objective and re-solves. The solution uses 10 truck trips and 14 EV trips, transporting 1,840 units. The original pollution objective remains 1,000; the augmented objective is 1,000.24.

**OUTCOME:** **Correct** (1000, matching reference)

**Analysis.** The initial solution is feasible and already minimizes pollution: at least 10 truck trips incur a lower bound of 1,000. Additional EV trips leave pollution unchanged. The edit therefore selects a less wasteful solution among pollution-optimal solutions, rather than correcting the primary objective value. In this instance, pollution values differ in multiples of 10, and the 24-trip solution has augmented value $1{,}000.24 < 1{,}010$; thus the weight 0.01 preserves the minimum pollution. A

small weight does not provide this guarantee in general; lexicographic optimization would explicitly preserve the primary objective.

Evaluation must use the original pollution objective (1,000), not the augmented objective (1,000.24). The trace illustrates a secondary preference for fewer trips that is absent from the original objective.

### D.4 Case Study 4: Strategic Reset After Accumulating Patches

**Input**

A 7-hub transportation network with bidirectional high-speed transit lanes. Each arc has a vehicle-per-hour capacity. The task is to find the maximum flow from Hub 0 (Central Dispatch) to Hub 6 (Main Distribution Hub).

**Agent Reasoning**

**Step 1.** The agent builds a max-flow LP. Solver returns objective $= 44$. Verification code computes source net outflow $= 25$. Discrepancy detected.
**Step 2.** The agent patches the model by introducing a `total_flow` auxiliary variable and rewriting source/sink constraints. New errors appear (return value mismatches, function signature conflicts).
**Step 3.** Over approximately 15 iterations, the agent produces 16 files (`model_correct.py`, `model_networkx.py`, `model_alternative.py`, `exhaustive_search.py`, etc.), exploring different fix strategies. Each patch introduces further inconsistencies.
**Step 4.** The agent prints: "The model has problems. Flow conservation constraints are not correctly applied. Let me simplify using the standard network flow formulation." It creates `model_simple.py` with a net-flow objective and explicit conservation constraints, obtaining 44 with successful verification. It then updates the submitted `model.py` and reruns that file, again obtaining 44 with capacity and conservation checks satisfied.

**Outcome:** **Correct** (44, matching reference)

**Analysis.** This case reveals an emergent "kill-and-restart" strategy. After accumulating patches that introduced diminishing returns and new bugs, the agent recognized that continuing to fix the existing code was less efficient than starting over with a clean design. This is a form of meta-reasoning where the agent evaluates not just the correctness of its model but also the productivity of its current debugging trajectory. The 16 intermediate files serve as exploration artifacts (some testing NetworkX-based algorithms, others trying exhaustive search) before the agent converges on the simplest viable formulation.

This illustrates that general-purpose agents can dynamically allocate their computational budget, exploring broadly when uncertain and resetting when accumulated complexity becomes counterproductive. A fixed pipeline has no mechanism for such strategic abandonment and restart.

### D.5 Case Study 5: Correct Diagnosis, Incorrect Fix

**Input**

This case uses the same problem as Case Study 3 and comes from iteration 2. The reference pollution optimum is 1,000.

**AGENT REASONING**

**Step 1.** The agent builds the same model as Case Study 3 with constraint $100x_1 + 80x_2 + 40x_3 + 60x_4 \geq 1800$ and objective min pollution. Solver returns pollution $= 1{,}000$ with 10 truck trips and 1,000 EV trips, transporting 61,000 units.
**Step 2.** The agent prints: "1,000 EV trips is clearly unreasonable, the same degenerate issue."
**Step 3.** The agent reasons: "Total transport far exceeds demand. Change $\geq 1800$ to $= 1800$ so transport exactly meets the requirement." Modifies the constraint.
**Step 4.** Re-solves. Result: trucks 10 (1,000 units) + EVs 12 (720 units) + motorcycles 2 (80 units) $= 1{,}800$ exactly. Pollution $= 1{,}020$. The agent accepts this answer.

**OUTCOME:** **Incorrect** (1,020 vs. reference 1,000)

**Analysis.** The true optimum is trucks $10$ + EVs $14 = 1{,}840$ units transported, pollution $= 1{,}000$. This solution exceeds $1{,}800$ by $40$ units but incurs zero additional pollution because EVs have zero marginal pollution cost. By imposing the equality $= 1{,}800$, the agent excluded this solution ($100 \times 10 + 60 \times 14 = 1{,}840 \neq 1{,}800$). Under the added equality, 10 truck trips, 12 EV trips, and 2 motorcycle trips form the least-polluting feasible solution, adding 20 units of avoidable pollution. Motorcycles are not the only way to meet demand exactly: 10 truck trips and 10 van trips also transport $1{,}800$ units, but incur pollution $1{,}500$.

The agent treated excess trips as a reason to replace a minimum-demand constraint with exact demand. This shrank the feasible region and excluded pollution-optimal solutions. If fewer trips are desired, that preference should be introduced without changing demand semantics, while preserving the primary pollution optimum.

The two traces distinguish an unnecessary change to feasibility from a secondary preference among optimal solutions. Iteration 2 raises pollution from $1{,}000$ to $1{,}020$; iteration 3 reduces trips while retaining pollution $1{,}000$. This contrast alone does not establish that attribution feedback caused the change in repair strategy.

## E COMPUTE RESOURCES

No local GPU is required: the local machine orchestrates API requests and executes generated optimization programs, while LLM inference runs through external APIs. Solver time depends on the formulation and instance and is subject to the execution limits in Appendix A.1. Under the schedule in Table 4, a full run with $T = 12$, $N = 5$, and $|\mathcal{D}| = 100$ entails $(1 + N)|\mathcal{D}|T = 7{,}200$ agent executions and $7{,}200$ judgment calls, together with $3T = 36$ attribution calls and $3NT = 180$ cascade-editing calls. These are scheduled counts, excluding initialization, retries, and any savings from caching or early termination; an agent execution may contain multiple API calls. Wall-clock time and API expenditure must be measured separately from this operation count.

## F LIMITATIONS

First, inference with the agent is expensive. Each test-time query invokes a general-purpose coding agent that may issue many tool calls before returning a model, making latency and cost noticeably higher than a single LLM call. Second, the agent's internal behavior is less interpretable than explicit multi-agent workflows, making it harder to attribute gains to planning, reflection, or coding.

## G BROADER IMPACT

OR-Skill may lower the barrier to applying operations research techniques in various domains by helping users translate natural-language descriptions into executable optimization models. However, LLMs may generate incorrect assumptions, biased objectives, or incomplete constraints, potentially leading to harmful decisions in high-stakes settings. Users may also overtrust executable code

and numerical outputs. Therefore, OR-Skill should be used as a modeling assistant rather than an autonomous decision-maker, and generated formulations should be reviewed by domain experts in consequential applications.