ORCAS: A CLAWS-STYLE MULTI-AGENT FRAMEWORK WITH ROUTED CORRECTION AND INSPECTABLE REPRESENTATION FOR OPTIMIZATION MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimization modeling is a long-horizon task that relies on professional knowledge and consists of multiple independent steps, making it one of the most critical yet technically demanding tasks in operations research (OR). Large language models (LLMs) have demonstrated immense potential in automated modeling across various scenarios. Existing agentic modeling methods, relying solely on step-by-step reasoning or prompts, are unable to overcome the difficulties of ineffective collaboration among agents and lack of persistent guidance, and often fail to achieve high performance. To address these challenges, we propose ORCAS, a CLAWs-style modeling system, representing modeling competence as an externalized, inspectable state that is routable, persistent, and improvable without fine-tuning. ORCAS adopts a multilayer architecture with agent-isolated workspaces and a routed double-loop correction mechanism that classifies each detected fault and routes it to the stage at which it could have arisen, thereby achieving separation of concerns and step-level provenance while enabling modeling capability to improve with use. Experiments on eight optimization modeling benchmarks show that ORCAS achieves an average accuracy of 79.5% and surpasses state-of-the-art agentic methods, with particularly large gains on challenging tasks.

1 INTRODUCTION

Optimization modeling translates real world decision problems into mathematical formulations and is central to applications such as resource allocation, scheduling, and transportation. Constructing a reliable model is a long-horizon task rather than a single act of translation. It requires domain expertise, mathematical abstraction, and a sequence of loosely coupled steps, including data extraction, variable definition, objective construction, constraint specification, in which an error introduced at any stage silently propagates to the final answer (Chen et al., 2023; 2024; Pu et al., 2026).

Recent advances in LLMs have driven rapid progress in automated optimization modeling, enabling models to generate mathematical formulations and executable solver code directly from natural language descriptions (Xiu et al., 2026; Li et al., 2026b; Zhao et al., 2026). Existing methods mainly follow two paradigms. Prompt-based methods provide modeling knowledge through search, retrieval, or agent collaboration (Xiu et al., 2026; Yuan et al., 2026), while fine-tuning methods incorporate such knowledge into model parameters through supervised or reinforcement learning (He et al., 2026; Zhao et al., 2026). Yet existing agentic modeling systems still struggle to overcome three persistent bottlenecks: they neither achieve genuine multi-agent division of labor, nor maintain persistent memory that survives across sessions, and lack step-level provenance. As an architecture that can only run once, it is difficult to further improve capabilities in long-horizon modeling tasks.

We therefore propose ORCAS (**O**perations **R**esearch **C**LAWs-style **A**gentic **S**ystem), a CLAWs-style (Karpathy, 2026; Willison, 2026) modeling system, which augments LLMs with long-term memory, autonomous planning, and multi-agent orchestration. Rather than eliciting knowledge transiently or encoding it in weights, ORCAS represents modeling competence as an externalized, inspectable state that is routable, persistable, and improvable, relocating learning into three coupled mechanisms. First, a multilayer architecture separates high-level for task planning, middle-level for formulation, execution, and verification, and bottom-level for memory, so that ORCAS could pro-

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.

vide step-level provenance and learn an inspectable representation of modeling competence that is external to the backbone and evolves with use. Second, a routed double-loop correction mechanism classifies each detected fault and routes it to the earliest representation at which it could have arisen, forming an inner code-repair loop and an outer replanning loop. Third, agent-isolated workspaces encapsulate each role's behavioral protocol, tools, role-specific skills, and accumulated memory, achieving true separation of concerns and persistent memory retention while converting transient conversations into verifiable files. On eight optimization modeling benchmarks, ORCAS achieves an average accuracy of 79.5% and surpasses state-of-the-art agentic methods, with the largest gains on the most challenging and heterogeneous tasks.

Our main contributions are as follows:

- We propose ORCAS, a CLAWs-style agentic framework for optimization modeling. Rather than eliciting knowledge transiently or fine-tuning, ORCAS provides step-level provenance and learns a persistent representation of modeling competence that evolves with use.
- We introduce a multilayer design in which four role-specialized agents operate over isolated workspaces and task representations, coupled with a routed double-loop correction mechanism, so that faults are localized and repaired at the earliest stage in which they arise.
- We design a modular skill mechanism that consolidates solving experience into verifiable files and reuses across sessions, enabling agents to work under the relevant guidance while remaining auditable.
- Experiments on eight optimization modeling benchmarks, together with extensive ablations, show that ORCAS improves over its backbone and is competitive with state-of-the-art baselines, demonstrating the effectiveness of the design. We further identify the dominant overlooked failure mode *false-positive infeasibility*.

2 Related Work

Large Language Models for Optimization Modeling Large language models have increasingly been used to translate natural-language problem descriptions into mathematical formulations and executable solver code. Existing approaches can be broadly divided into prompt-based and learning-based methods.Prompt-based methods keep the backbone model fixed and improve modeling ability through inference-time scaffolding. Chain-of-Experts (Xiao et al., 2024) coordinates role-specialized experts to formulate and solve complex operations research problems, while OptiMUS (AhmadiTeshnizi et al., 2024) adopts a modular workflow that constructs mathematical models, generates solver code, and iteratively repairs modeling and execution errors. OptiTree (Liu et al., 2025) further replaces fixed-step decomposition with a hierarchical tree search that retrieves and combines modeling thoughts from simpler subproblems. In parallel, learning-based methods internalize optimization knowledge through domain-specific training. ORLM (Huang et al., 2025a) introduces OR-Instruct for training open-source optimization models, LLMOPT (Jiang et al., 2025) uses a five-element formulation and multi-instruction tuning, and OptMATH (Lu et al., 2025) develops a bidirectional synthesis framework for scalable, solver-validated training data. More recent methods incorporate solver-grounded reinforcement learning, like SIRL (Chen et al., 2026a), using solver-informed rewards, and StepORLM (Zhou et al., 2026), which co-evolves a policy model and a generative process verifier. OR-R1 (Ding et al., 2025) combines supervised fine-tuning with test-time group-relative policy optimization to improve data efficiency. Most recently, COOPA (Li et al., 2026a) combines three components: iterative , element-level provenance and confidence explanations, and multi-solver routing to specialized optimizer agents for different OR problem classes.

Agentic Self-Correction and Experience Memory A growing body of research equips LLM agents with feedback loops and external memory so that they can improve without updating LLMs parameters. Early inference-time methods mainly revise the current solution. Self-Refine (Madaan et al., 2023) alternates between self-generated feedback and output refinement, whereas Reflexion (Shinn et al., 2023) converts environmental feedback into verbal reflections and stores them in an episodic memory buffer for subsequent trials. Later work extends this idea from within-task correction to cross-task experience transfer. ExpeL (Zhao et al., 2024) extracts reusable natural-language

insights from successful and failed trajectories, and Agent Workflow Memory (Wang et al., 2025) induces recurring workflows from prior demonstrations or online interactions to guide future actions. More recent systems organize experience into increasingly structured and updateable memory units. A-MEM (Xu et al., 2025) dynamically creates and links memories through an agentic note-taking process, SkillWeaver (Zheng et al., 2025) converts successful interactions into reusable executable skills, and ReasoningBank (Ouyang et al., 2026) abstracts both successful and failed trajectories into transferable reasoning strategies. These principles have also begun to influence optimization modeling, like MIRROR (Shi et al., 2026), AlphaOPT and Agora-Opt (Lin et al., 2026). Together, these studies demonstrate a broader transition from transient self-correction toward agents that retain, organize, and reuse experience across tasks. Despite this progress, most systems implement collaboration, correction, and experience reuse as ephemeral processes over shared conversational state, which cannot be audited, routed, or persisted across sessions. ORCAS addresses this gap through a CLAWs-style architecture that externalizes every intermediate output as an inspectable file-based representation, providing step-level provenance and enabling each detected fault to be routed to the earliest stage at which they arose.

3 ORCAS: Persistent Agentic Optimization Modeling

Let $p \in \mathcal{P}$ denote a natural-language OR problem. A valid output consists of a mathematical formulation m, an executable solver program c, and a solution $y = (\mathbf{x}^\star, f^\star, s)$, where $x^\star$ is the returned decision vector, $f^\star$ is the reported objective value, and s is the solver status. The objective of ORCAS is not merely to generate a solution, but to construct a verified transformation

$$\mathcal{F} : p \longmapsto (m, c, y). \tag{1}$$

In this section, we introduce ORCAS, a persistent agentic framework with long-term memory and autonomous planning capabilities for optimization modeling. Rather than updating the parameters of the underlying LLMs, ORCAS enhances modeling capabilities through three coupled mechanisms, as shown in Figure 1: (i) a multilayer architecture with a high-level layer for task planning, a middle-level layer for formulation, execution, and verification, and a bottom-level layer for the memory mechanism; (ii) a routed feedback mechanism consisting of an inner code-repair loop and an outer modeling-replanning loop; (iii) agent-isolated workspaces that encapsulate behavioral protocols, tools, role-specific skills, and memory.

3.1 Multilayer Modeling and Memory Architecture

ORCAS decomposes optimization modeling into three logical levels rather than one or several agents in sequence to maintain the entire trajectory. The high level uses the Planner to focus on task decomposition and execution order, without committing to mathematical expressions. The middle level utilizes the Modeler, Coder and Reviewer to transform natural-language semantics into a rigorous formulation and an executable code, and decide whether the modeling chain is acceptable. The bottom level relies on the CLAWs memory mechanism to summarize and maintain the experience of specific roles during the modeling process. Given problem p, the forward pass at cycle k is

$$\begin{aligned} z_P^k &= F_P(p; \mathcal{W}_P), \\ z_M^k &= F_M(p, z_P^k; \mathcal{W}_M), \\ z_C^k &= F_C(z_M^k; \mathcal{W}_C), \\ z_R^k &= F_R(p, z_P^k, z_M^k, z_C^k; \mathcal{W}_R), \end{aligned} \tag{2}$$

where z_P^k is a structured workflow, z_M^k is a source-traced mathematical model, $z_C^k = (c^k, o^k)$ contains solver code and its execution trace, and z_R^k contains the review verdict, diagnosed root cause, repair target, and correction instructions. Typed interfaces make each stage independently inspectable and prevent downstream agents from relying on an unstructured shared dialogue.

High-level planning. The Planner first classifies the problem type and produces an ordered workflow $z_P^k = (\tau_1, \ldots, \tau_L)$, where each step specifies what to do, required input, expected output, success criterion, and problem-specific precaution. The workflow covers modeling related steps such

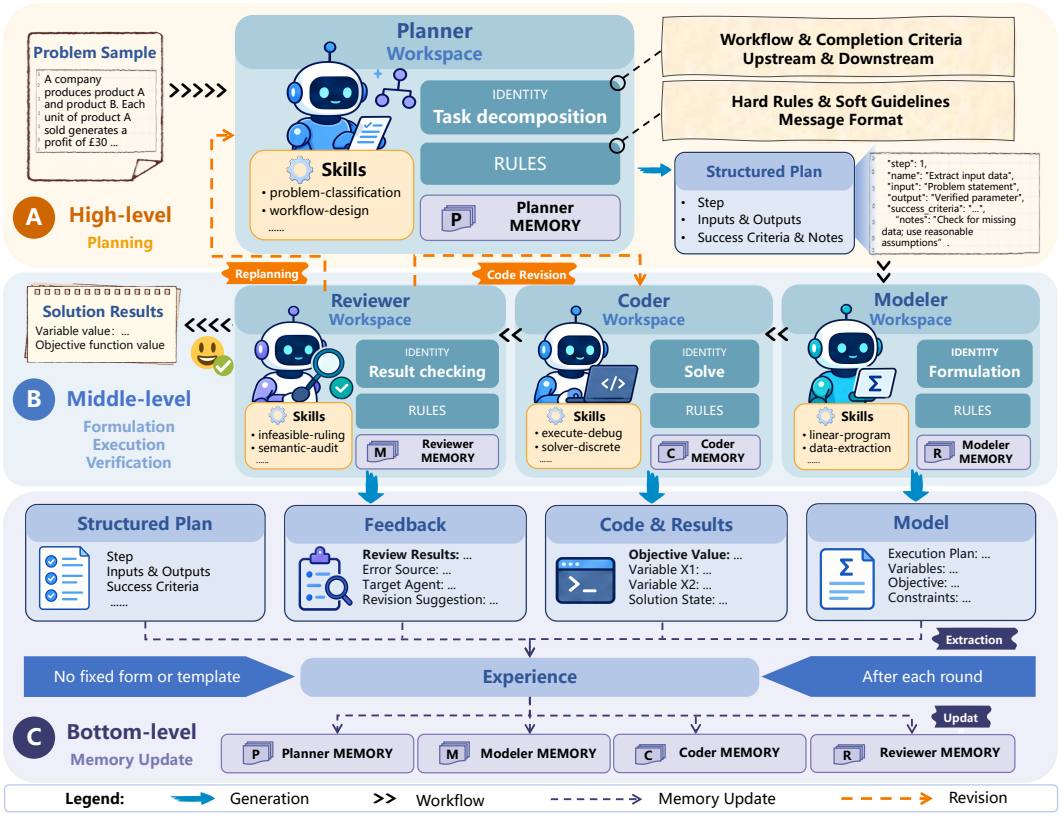


Figure 1: The overview of ORCAS. The Planner decomposes a natural-language problem into a structured stepwise plan. The Modeler formulates the math model, the Coder solves it, and the Reviewer audits the results. An inner loop feeds program revisions back to Coder, while an outer loop escalates unreasonable results back to Planner. Every agent works in an isolated workspace with its own skills and long-term memory that updates with use.

as data extraction, decision-variable definition, and solver selection, minimizing the involvement of specific mathematical model. The purpose of this design is to separate task decomposition and execution order from formulation details, thereby reducing premature mathematical commitments and giving downstream agents an explicit completion contract.

Middle-level formulation. The Modeler executes the workflow in order and maps the problem semantics to an implementation-ready model

$$z_M^k = \left(\mathcal{X}^k, f^k, \mathcal{G}^k, \Theta^k, \mathcal{U}^k\right), \tag{3}$$

where $\mathcal{X}^k$ defines the decision variables and domains, f^k is the objective, $\mathcal{G}^k$ is the constraint set, Θ^k contains parameters, and $\mathcal{U}^k$ records explicit assumptions. Every coefficient and constraint are linked to a sentence in p or to a documented derivation, retaining enough provenance to expose semantic omissions before they are hidden in the code.

Middle-level execution. The Coder translates z_M^k into a complete solver program c^k, selects a solver compatible with the model class, executes the program in an isolated sandbox, and records the execution trace, making the progress operational and reproducible

$$o^k = (s^k, \mathbf{x}^k, \hat{f}^k, \mathbf{e}^k, \Delta t^k), \tag{4}$$

where s^k is the solver status, $\mathbf{x}^k$ returns variable assignment, $\hat{f}^k$ reports objective, $\mathbf{e}^k$ stores the standard-error trace, and Δt^k is the runtime.

Middle-level verification. The Reviewer acts as an adversarial quality gate rather than a passive evaluator. It reads the original problem together with the current workflow, formulation, solver program, and execution trace, and performs an ordered four-layer audit.

- Problem-plan alignment V^k_{sem}, including objective direction, problem variant, explicit and implicit data, step dependencies, modeling completeness, and the operability of each step.
- Model fidelity against the original problem V^k_{form}, including the definitions and domains of variables, parameter provenance, objective semantics, constraint coverage and direction, logical structure, dimensional consistency, and any assumptions or derived data.
- Model-code fidelity V^k_{code}, like one-to-one mappings of variables and parameters, variable types and bounds, index ranges, objective direction and coefficients, constraint encodings, solver invocation, and answer extraction.
- Solver status and result plausibility V^k_{phys}, including integrality, signs and natural bounds, unit and scale consistency, structural implementability, and known anomaly signatures, for example, the result of a variable representing a certain number is negative, or the target value exceeds the upper limit.

In parallel with this semantic audit, reviewer will also perform backtesting of the results, which involves substituting variable values into constraint conditions to determine if the solution is compliant. For each parseable constraint $g_j(\mathbf{x}) \bowtie_j b_j$, it computes

$$V^k_j = \mathbb{I}\big[g_j(\mathbf{x}^k) \bowtie_j b_j\big]. \tag{5}$$

The final acceptance gate is

$$A^k = V^k_{\text{sem}} \wedge V^k_{\text{form}} \wedge V^k_{\text{code}} \wedge \left(\bigwedge_j V^k_j\right) \wedge V^k_{\text{phys}}. \tag{6}$$

The Reviewer issues the strict verdict APPROVED only when $A^k = 1$; any failed gate produces CHANGES_NEEDED together with the root cause, repair target, actionable instructions, and diagnostic questions tied to the original problem. Pure implementation defects are routed to the Coder, whereas errors in task interpretation, data extraction, or mathematical formulation are routed to the Planner, with a Modeler-specific correction attached when the plan is correct but its mathematical derivation is not. On every repair cycle, the Reviewer rereads and re-audits all updated artifacts.

Bottom-level memory and persistence. The CLAWs memory layer operates alongside the forward pass, so modeling and experience preservation occur within the same process. ORCAS maintains persistence at two complementary levels. The first is an append-only trajectory store

$$\mathcal{H}_t = \mathcal{H}_{t-1} \cup \left\{z^k_P, z^k_M, c^k, o^k, z^k_R\right\}_{k=1}^{K_t}, \tag{7}$$

where each artifact is indexed by problem and correction cycle. This store is updated as the agents model, execute, and review the problem rather than only after completion.

The second is role-specific memory, which is learned by the system itself, not pre-set. Agent i converts relevant trajectory evidence into a compact experience e^t_i and updates its local memory $\mathcal{M}^{t+1}_i$ after task t. In the next task, the relevant memory will be loaded, which constitutes architectural self-evolution rather than parameter learning.

The persistence distinguishes the two timescales of ORCAS: fast correction serves for the current model, while slower, conditional memory consolidation improves the external knowledge available to future tasks. Separation between the persistence of the raw trajectory and the consolidated experience also prevents every intermediate message from being treated as trusted long-term knowledge.

3.2 ROUTED DOUBLE-LOOP CORRECTION

When $A^k = 0$, the Reviewer produces structured feedback

$$z^k_R = (\rho^k, d^k, q^k, u^k), \tag{8}$$

where ρ^k denotes the diagnosed root-cause class, d^k identifies failed checks, q^k contains diagnostic questions, and u^k specifies an executable repair instruction.

ORCAS routes this feedback according to the earliest representation at which the defect could have arisen. The first branch forms the *inner model–code loop*. Syntax errors, solver-API errors, variable-type mismatches in code, and incorrect result extraction return directly to the Coder, retaining the accepted workflow and model, which could repair local implementation defects with minimal recomputation and without destabilizing validated upstream semantics. The second branch forms the *outer result–planning loop*. A semantic reversal, omitted constraint family, incomplete data extraction, or physically implausible solution are returned to the Planner, and the modified workflow is propagated through downstream agents, allowing for deep rollback. Together, rooted correction avoids regenerating verified upstream artifacts for a local implementation error while still revisiting the earliest defective representation when necessary.

3.3 Independent Agent Workspaces and Role-Specific Skills

We define the agent set as $\mathcal{A} = \{P, M, C, R\}$, corresponding to the Planner, Modeler, Coder, and Reviewer. For agent $i \in \mathcal{A}$, let $\mathcal{W}_i^k$ be its private workspace at cycle k,

$$\mathcal{W}_i = \left(q_i^k, \mathcal{I}_i^k, \mathcal{S}_i(p), \mathcal{M}_i^t\right), \tag{9}$$

where q_i^k is the role instruction, $\mathcal{I}_i^k$ is the projection of the global task state, $\mathcal{S}_i$ is skill library, and $\mathcal{M}_i^t$ is persistent memory before task t. Equation 9 emphasizes two design choices: agents receive only the information required by their roles, and reusable skills and memories remain external to model parameters. This separation also makes individual roles easier to maintain or update without redesigning the complete pipeline.

Skills encode reusable OR procedures as inspectable modules instead of repeatedly eliciting them from LLMs reasoning, which converts recurring failure-prone judgments, such as objective direction, integrality, big-M construction, dimensional consistency, or infeasibility diagnosis, into explicit procedures that can be inspected and ablated. At the same time, stable domain knowledge is separated from task specific context, meaning that enhancing professional competence only requires updating skills in the corresponding workspace, without modifying the prompts or the architecture. For an incoming problem, each agent activates a task-dependent subset from the whole skills library.The complete skills design can be found in the Appendix D.2.

4 Experiments

4.1 Experimental Setup

Benchmarks. We evaluate ORCAS on eight optimization modeling benchmarks: NL4Opt (Ramamonjison et al., 2023), MAMO EasyLP (Huang et al., 2025b), MAMO ComplexLP (Huang et al., 2025b), NLP4LP (AhmadiTeshnizi et al., 2024), ComplexOR (Xiao et al., 2024), IndustryOR (Huang et al., 2025a), OptiBench (Yang et al., 2025), and OptMATH (Lu et al., 2025). These benchmarks span diverse optimization modeling tasks and vary substantially in problem scale, structural complexity, and domain context, ranging from relatively standard linear programs to industrial and combinatorial optimization problems. More details about benchmarks are provided in Appendix A.

Baselines. We compare ORCAS with three groups of methods: general LLMs, fine-tuned models and agentic methods. General LLMs include GPT-5 (OpenAI, 2026), DeepSeek-V4 (DeepSeek-AI, 2026), DeepSeek-R1 (DeepSeek-AI, 2025) and Qwen2.5-72B (Qwen Team, 2024). Fine-tuned baselines include ORLM-LLaMA3-8B (Huang et al., 2025a), LLMOPT-Qwen2.5-14B (Jiang et al., 2025), OptMATH-Qwen2.5-7B (Lu et al., 2025), SIRL-Qwen2.5-7B (Chen et al., 2026b), and StepORLM-Qwen3-8B (Zhou et al., 2026). Agentic baselines include CoE (Xiao et al., 2024) and OptiMUS (AhmadiTeshnizi et al., 2024), which rely on multi-agent collaboration; OptiTree (Liu et al., 2025) and AlphaOPT (Kong et al., 2025), which further incorporate external OR knowledge such as hierarchical modeling-thought trees or solver-verified experience libraries; and ReLoop (Lian et al., 2026), which integrates structured modeling with behavioral verification. Further baseline descriptions and configurations are provided in Appendix A.

Table 1: The overall performance with Pass@1 accuracy (%). Best results are in bold and second-best results are underlined. Missing entries are denoted by -.

Method	NL4Opt	MAMO Easy	MAMO Cpx	NLP4LP	CpxOR	IndOR	OptM.	OptiB.
General LLMs								
GPT-5	71.9	92.7	48.6	87.6	61.1	43.0	35.5	76.4
DeepSeek-V4	78.4	93.6	52.3	92.1	55.6	45.0	39.8	78.6
DeepSeek-R1	76.1	89.7	48.6	90.4	55.6	45.0	34.9	67.2
Qwen2.5-72B	74.6	94.8	41.4	88.2	50.0	38.0	15.1	54.6
Fine-tuned Models								
ORLM	73.8	90.4	59.5	76.4	50.0	43.0	2.6	61.8
LLMOPT	80.3	89.5	44.1	73.4	35.3	29.0	12.5	53.8
OptMATH	95.9	89.9	54.1	-	-	31.0	34.7	57.9
SIRL	94.8	94.6	61.1	96.6	44.4	48.0	45.8	81.6
StepORLM	96.7	95.4	77.5	97.2	50.0	52.4	40.4	80.6
Agentic Methods								
CoE	78.9	86.1	57.7	87.4	50.0	33.0	39.2	82.6
OptiMUS	80.6	88.4	60.2	88.2	44.4	36.0	41.3	83.1
AlphaOPT	93.0	92.5	62.1	90.4	54.3	49.0	39.2	80.6
OptiTree	**97.5**	**95.8**	**81.5**	93.6	50.0	54.0	52.4	<u>86.9</u>
ReLoop	81.2	89.6	62.4	79.9	<u>55.6</u>	36.0	27.1	79.9
ORCAS(0)	96.3	94.3	<u>78.2</u>	<u>97.7</u>	**66.7**	<u>55.0</u>	<u>54.4</u>	85.9
ORCAS(200)	<u>96.7</u>	<u>95.2</u>	<u>78.2</u>	**98.9**	**66.7**	**57.0**	**55.4**	**87.3**

Implementation and Evaluation. We report solving accuracy on each benchmark. Generated solutions are evaluated through solver execution and objective matching under a unified evaluation protocol. For agentic methods, DeepSeek-V4 is used as the backbone whenever applicable. Complete evaluation protocols and implementation details are provided in Appendix A.

4.2 Main Results

We first evaluate the effectiveness of ORCAS(0), the base framework without memory accumulation, and ORCAS(200), which has accumulated memory from 200 OR-Instruct (Huang et al., 2025a) problems, against three families of baselines on eight challenging benchmarks with DeepSeek V4. The results, presented in Table 1, highlight two key findings.

High Accuracy. ORCAS(200) attains the highest accuracy on five of the eight benchmarks and the highest average accuracy overall (79.5%), exceeding the strongest agentic baseline OptiTree (76.5%) by 3.0 points. The advantage is concentrated on benchmarks with more complex modeling structures, including ComplexOR, IndustryOR, and OptMATH, on which ORCAS(200) averages 59.7%, versus 52.1% for OptiTree and 47.6% for StepORLM, a margin of 7.6 points. Performance on relatively saturated tasks remains competitive. At the same time, ORCAS(0) already reaches state-of-the-art accuracy, with an average of 78.5%, confirming that the CLAWs paradigm, routed correction, and workspace isolation account for the majority of the improvement. A blinded evaluation of 200 intermediate models by five OR researchers further confirms that ORCAS produces structurally complete formulations.

Generalization of Memory. Comparing the memory-free ORCAS(0) with ORCAS(200) isolates the contribution of the bottom-level memory mechanism from that of the multilayer architecture. Accumulating experience on 200 non-test problems raises the average to 79.5% (+1.0 percentage point). The gains are unevenly distributed and follow benchmark composition: they are largest on IndustryOR ($+2.0$), OptiBench ($+1.4$), NLP4LP ($+1.2$), and OptMATH ($+1.0$), which contain diverse problem classes, and vanish on ComplexOR and MAMOComplexLP, whose small test sizes make the accuracy granularity coarser than the observed effect. The McNemar's test confirms that the improvement from persistent memory is statistically reliable rather than an artifact of benchmark

Table 2: Ablation of the multilayer architecture. Accuracy (%); Δ is the change relative to the full system.

Configuration	NL4OPT	MAMOEasyLP	IndustryOR	OptMATH	Mean	Mean Δ
w/o Planner	93.9	95.0	45.0	43.5	69.4	-6.7
w/o Coder	91.9	92.5	47.0	44.6	69.0	-7.1
w/o Reviewer	69.7	90.0	45.0	34.8	59.9	-16.2
w/o Memory	96.3	94.3	55.0	54.4	75.0	-1.1
ORCAS(200), full	**96.7**	**95.2**	**57.0**	**55.4**	**76.1**	—

composition, which are reported in detail in Appendix B.3. Memory therefore acts as an incremental refinement over a strong architecture, making the system more powerful as it is used.

Beyond aggregate accuracy, the complete outputs reveal a failure mode that has been largely overlooked in prior work: *false-positive infeasibility*. No residual error is attributable to non-executable code, unhandled exceptions, or solver timeouts; instead, the single largest error class is a formulation that declares the problem `INFEASIBLE` even though the data required to construct a feasible model are fully recoverable from the problem statement, accounting for 41.3% of all incorrect instances. A detailed per-benchmark statistics are provided in Appendix B.10. To further demonstrate the robustness of ORCAS, Appendix B reports additional studies, including a McNemar test, data source and size for memory construction, memory form comparison, backbone generalization across three LLMs, and comparison with simple iterative prompting under the same inference budget.

4.3 Ablation Studies

Running every ablation on all eight benchmarks is computationally prohibitive. To comprehensively assess the contribution of each architectural component, we select four representative benchmarks, including NL4OPT, MAMO-EasyLP, IndustryOR and OptMATH, for the ablation study. These benchmarks are chosen to span diverse difficulty levels, problem types, and application scenarios.

Impact of the Multilayer Architecture To systematically evaluate the contribution of each component in the multilayer architecture, we design a dedicated ablation study along three dimensions with modeling accuracy as the primary evaluation metric. (1) At the high level, ORCAS without Planner requires the Modeler to formulate the mathematical model directly from the raw problem description, without explicit stepwise planning. (2) At the middle level, ORCAS without Coder requires the Modeler to generate both the formulation and the executable solver code, while ORCAS without Reviewer removes the verification and feedback loop so that the system directly outputs the generated solution without iterative inspection. (3) For the bottom-level memory mechanism, we reuse the ORCAS(0) versus ORCAS(200) comparison reported in the main experiments when both configurations share the same frozen memory protocol and evaluation pipeline. As shown in Table 2, removing any component leads to a consistent decrease in mean accuracy, demonstrating the effectiveness of the multilayer architecture.

Impact of Routed Correction To evaluate the effectiveness of the routed correction mechanism, we conduct a dedicated ablation study that examines how routing detected faults to the appropriate agent affects modeling performance. In the full ORCAS, the Reviewer classifies each identified fault and routes it to either the Planner or the Coder according to its nature, enabling targeted repair at the correct level. We compare this full system against three ablated variants. (1) All detected faults are routed exclusively to the Planner for revision, regardless of whether they originate from planning or coding. (2) All faults are routed exclusively to the Coder for repair, preventing the Planner from revisiting the task decomposition. (3) The verification and feedback loop is removed entirely, and the system directly outputs the generated model and code without any correction. As shown in Figure 2 (a), different routing strategies produce different effects, but our designed routed double-loop correction yields the best results.

Impact of Independent Workspaces ORCAS assigns each role an independent workspace containing only that role's operating instructions, skill library, and accumulated experience. We ablate

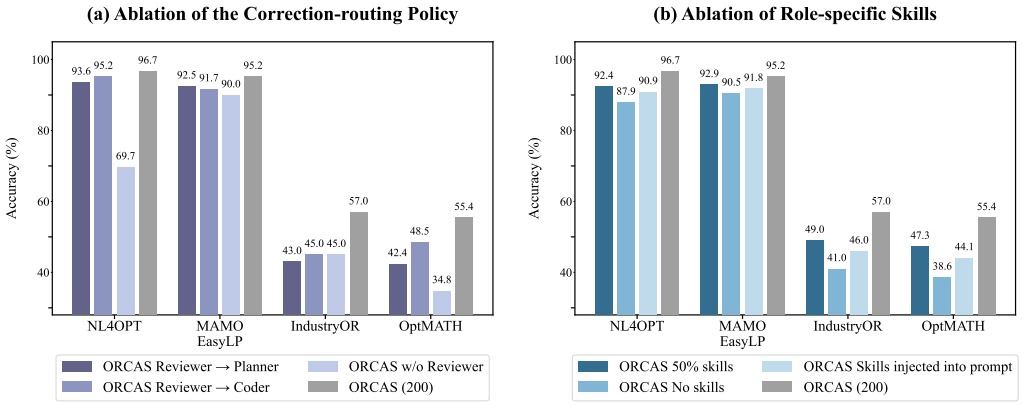


Figure 2: Ablation studies of Routed Correction and Role-Specific Skills. (a) Performance under different correction-routing policy. (b) Performance under different skill levels.

Table 3: Ablation of workspace and context isolation. Accuracy (%).

Configuration	NL4OPT	MAMO EasyLP	IndOR	OptM.	Mean	Mean Δ
Shared workspace & Independent contexts	84.9	90.4	51.1	49.2	68.9	−7.2
Shared workspace & Shared context	78.6	87.4	50.0	42.9	64.7	−11.4
ORCAS(200), isolated	**96.7**	**95.2**	**57.0**	**55.4**	**76.1**	—

this design along two nested levels. On the agent platform used here, an isolated workspace necessarily implies an isolated conversational context, so the configuration (isolated workspace, shared context) is not realizable; the ablation is consequently *nested* rather than fully factorial. (1) Shared workspace & independent contexts: A single shared workspace merges all role documents and skill libraries. Four agents are instantiated within it, one per role, each with its own conversation. (2) Shared workspace & shared context: A single shared workspace, as described above, but a single agent executes all four roles sequentially within one conversation, changing role according to task progress. As shown in Table 3, merging workspaces into a shared one degrades accuracy even when contexts remain separate and collapsing the contexts compounds the loss, thereby demonstrating the effectiveness of the independent workspace.

Impact of Role-Specific Skills The workspace of each agent contains a set of procedural skill documents, retrieved on demand according to the problem type rather than loaded unconditionally. We ablate both the *quantity* of available skills and the *mechanism* by which they enter the model's context. (1) 50% skills: Half of each role's skill documents are randomly removed, with the retained subset held fixed. (2) No skills: All skill documents are removed. (3) Skills injected into the prompt: All skills contents are concatenated into the agent prompt and loaded unconditionally for every problem, replacing on-demand retrieval. Figure 2 (b) shows that the accuracy decreases monotonically with skill availability, and unconditional prompt injection is less stable than on-demand retrieval, demonstrating the effectiveness of the role-specific skills.

5 Conclusion

We present ORCAS, a CLAWs-style modeling system that adopts a multi-layer architecture, iterative feedback loops, agent-isolated workspaces, and modular skill design to enhance long-horizon task-solving capability. ORCAS improves modeling performance in prompt-based settings without parameter tuning. In future work, the intermediate outputs—including plans, mathematical models, and feedback—can support both external experience reuse and further capability improvement through fine-tuning.

AI USE STATEMENT

In this work, we used generative AI tools for translation and language polishing to improve the grammar, clarity, and readability of the manuscript. We did not use generative AI for other tasks requiring disclosure, such as developing research methods, formulating scientific claims, designing experiments, implementing the proposed methods, or interpreting results. All AI-assisted content was reviewed by the authors, who take full responsibility for the final manuscript.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. Optimus: Scalable optimization modeling with (mi) lp solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 577–596, 2024.

Yitian Chen, Cheng Cheng, Yinan Sun, Zi Ling, and Dongdong Ge. Opt-engine: Benchmarking the limits of llms in optimization modeling via complexity scaling. In *Forty-third International Conference on Machine Learning*, 2026a.

Yitian Chen, Jingfan Xia, Siyu Shao, Dongdong Ge, and Yinyu Ye. Solver-informed rl: Grounding large language models for authentic optimization modeling. *Advances in Neural Information Processing Systems*, 38:106027–106069, 2026b.

Ziang Chen, Jialin Liu, Xinshang Wang, and Wotao Yin. On representing mixed-integer linear programs by graph neural networks. In *The Eleventh International Conference on Learning Representations*, 2023.

Ziang Chen, Jialin Liu, Xiaohan Chen, Xinshang Wang, and Wotao Yin. Rethinking the capacity of graph neural networks for branching strategy. *Advances in Neural Information Processing Systems*, 37:123991–124024, 2024.

DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025.

DeepSeek-AI. DeepSeek-V4 technical report, 2026.

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. *arXiv preprint arXiv:2511.09092*, 2025. URL https://arxiv.org/abs/2511.09092.

Yiliu He, Tianle Li, Binghao Ji, Zhiyuan Liu, and Di Huang. Evoopt-llm: Evolving industrial optimization models with large language models. *arXiv preprint arXiv:2602.01082*, 2026. URL https://arxiv.org/abs/2602.01082.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 2025a. doi: 10.1287/opre.2024.1233. URL https://doi.org/10.1287/opre.2024.1233.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. Llms for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710, 2025b.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *Proceedings of the International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=YHajCuL9qW.

Andrej Karpathy. Just like LLM agents were a new layer on top of LLMs, Claws are now a new layer on top of LLM agents, taking the orchestration, scheduling, context, tool calls and a kind of persistence to a next level, feb 2026. URL https://twitter.com/karpathy/status/2024987174077432126. Tweet.

Minwei Kong, Ao Qu, Xiaotong Guo, Wenbin Ouyang, Chonghe Jiang, Han Zheng, Yining Ma, Dingyi Zhuang, Yuhan Tang, Junyi Li, Shenhao Wang, Haris Koutsopoulos, Hai Wang, Cathy Wu, and Jinhua Zhao. AlphaOPT: Formulating optimization programs with self-improving LLM experience library. *arXiv preprint arXiv:2510.18428*, 2025. URL https://arxiv.org/abs/2510.18428.

Chuanhao Li, Xiaoan Xu, Dirk Bergemann, Ethan X. Fang, Yehua Wei, and Zhuoran Yang. Coopa: A modular llm agent architecture for operations research problems, 2026a. URL https://arxiv.org/abs/2606.27611.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. In *Forty-third International Conference on Machine Learning*, 2026b.

Junbo Jacob Lian, Yujun Sun, Huiling Chen, Chaoyu Zhang, Hanzhang Qin, and Chung-Piaw Teo. ReLoop: Structured modeling and behavioral verification for reliable LLM-based optimization. *arXiv preprint arXiv:2602.15983*, 2026. URL https://arxiv.org/abs/2602.15983.

Jianghao Lin, Zi Ling, Chenyu Zhou, Tianyi Xu, Ruoqing Jiang, Zizhuo Wang, and Dongdong Ge. From soliloquy to agora: Memory-enhanced LLM agents with decentralized debate for optimization modeling. *arXiv preprint arXiv:2604.25847*, 2026. URL https://arxiv.org/abs/2604.25847.

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 *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/aeaf9b454f4670d5bf2f02b4b6e5bffd-Abstract-Conference.html.

Hongliang Lu, Zhonglin Xie, Yaoyu Wu, Can Ren, Yuxuan Chen, and Zaiwen Wen. OptMATH: A scalable bidirectional data synthesis framework for optimization modeling. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 40769–40802. PMLR, 2025. URL https://proceedings.mlr.press/v267/lu25o.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

OpenAI. Openai gpt-5 system card, 2026. URL https://arxiv.org/abs/2601.03267.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *Proceedings of the International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/980ea04d23d1f6908964eba2a74afe45-Abstract-Conference.html.

Tianle Pu, Jianing Li, Yingying Gao, Shixuan Liu, Zijie Geng, Haoyang Liu, Chao Chen, and Changjun Fan. Coco-milp: Inter-variable contrastive and intra-constraint competitive milp solution prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 24882–24890, 2026.

Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. URL https://arxiv.org/abs/2412.15115.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, et al. Nl4opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS 2022 competition track*, pp. 189–203. PMLR, 2023.

Yifan Shi, Jiayi Wang, Minyi Wu, Ye Fan, Jialong Shi, and Jianyong Sun. MIRROR: A multi-agent framework with iterative adaptive revision and hierarchical retrieval for optimization modeling in operations research. *arXiv preprint arXiv:2602.03318*, 2026. URL https://arxiv.org/abs/2602.03318.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025. URL https://proceedings.mlr.press/v267/wang25bx.html.

Simon Willison. Andrej Karpathy talks about "Claws", feb 2026. URL https://simonwillison.net/2026/Feb/21/claws/. Blog post.

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 *Proceedings of the International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=HobyL1B9CZ.

Xianchao Xiu, Jianhao Li, Jun Fan, and Wanquan Liu. Large language models for operations research: A comprehensive survey. *arXiv preprint arXiv:2605.20849*, 2026. URL https://arxiv.org/abs/2605.20849.

Wujiang Xu, Kai Mei, Hang Gao, Juntao Tan, Zujie Liang, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.

Zhicheng Yang, Yiwei Wang, Yinya Huang, Zhijiang Guo, Shi Shi, Xiongwei Han, Liang Feng, Linqi Song, Xiaodan Liang, and Jing Tang. Optibench meets resocratic: Measure and improve llms for optimization modeling. In *International Conference on Learning Representations*, volume 2025, pp. 24726–24759, 2025.

Haofeng Yuan, Jianing Peng, Jieyi Bi, Ni Zhang, Shiji Song, and Zhiguang Cao. Formuevo: Llm-guided evolution for discovering solver-efficient mixed-integer programming formulations. *arXiv preprint arXiv:2608.23353*, 2026. URL https://arxiv.org/abs/2608.23353.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 2024. URL https://ojs.aaai.org/index.php/AAAI/article/view/29936.

Ruiqing Zhao, Fengzhi Li, Yuan Zuo, Rui Liu, Yansong Liu, Yunfei Ma, Fanyu Meng, and Junlan Feng. Strategy-aware optimization modeling with reasoning llms. In *Forty-third International Conference on Machine Learning*, 2026.

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

Chenyu Zhou, Tianyi Xu, Jianghao Lin, and Dongdong Ge. StepORLM: A self-evolving framework with generative process supervision for operations research language models. In *Proceedings of the International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/0bcfb525c8f8f07ae10a93d0b2a40e00-Abstract-Conference.html.

A Experimental Details

A.1 Datasets and Evaluation Protocol

Benchmark Statistics and Preprocessing. We evaluate ORCAS on eight established optimization modeling benchmarks: NL4Opt (Ramamonjison et al., 2023), MAMO EasyLP (Huang et al., 2025b), MAMO ComplexLP (Huang et al., 2025b), NLP4LP (AhmadiTeshnizi et al., 2024), ComplexOR (Xiao et al., 2024), IndustryOR (Huang et al., 2025a), OptiBench (Yang et al., 2025), and OptMATH (Lu et al., 2025). Together, these benchmarks cover a broad spectrum of optimization modeling difficulty, ranging from relatively standard linear programs to problems involving more complex constraint interactions, logical dependencies, integer decisions, and application-specific modeling structures.

NL4Opt originates from the NeurIPS optimization modeling competition and primarily contains relatively compact linear programming problems. MAMO provides two evaluation sets with different levels of modeling complexity: EasyLP contains comparatively direct linear programming instances, whereas ComplexLP contains problems with more involved constraint structures and reasoning requirements. NLP4LP provides human-written optimization problems with reference solutions and executable solver programs. Its instances focus on classical LP, MILP, and MINLP modeling scenarios, with relatively explicit problem statements. ComplexOR consists of challenging OR modeling problems collected from complex application settings. The problems require nontrivial formulation decisions, including variable design, constraint organization, and mixed-integer reasoning. IndustryOR consists of optimization problems derived from industrial and application-oriented scenarios. OptMATH contains mathematically structured optimization problems and places greater emphasis on complex formulation and combinatorial reasoning. OptiBench evaluates optimization modeling across a broader collection of problem structures and was designed to provide a more challenging assessment of model formulation ability. Table 4 summarizes the evaluation sets used in our experiments.

Table 4: Statistics of the optimization modeling benchmarks used for evaluation. The number of instances corresponds to the evaluation files used in our experiments.

Benchmark	# Instances	Characteristics
NL4Opt	245	Standard linear programming
MAMO EasyLP	545	Relatively simple LP modeling
MAMO ComplexLP	211	Complex and coupled LP modeling
NLP4LP	178	LP, MILP and MINLP
ComplexOR	18	complex LP and MILP
IndustryOR	100	Application-oriented industrial OR
OptMATH	193	Mathematical and combinatorial modeling
OptiBench	403	Diverse optimization modeling problems
Total	1,893	

We use the released English problem descriptions and reference targets and normalize the benchmark records into a common evaluation format. For the benchmark instances for which corrected reference targets are available, we use the corrected versions throughout our in-house evaluation. These corrections affect only the reference targets and leave the natural-language problem descriptions unchanged. Specifically, the corrected evaluation files revise 2 instances in NL4Opt, 14 instances in MAMO EasyLP, and 24 instances in MAMO ComplexLP, while NLP4LP, ComplexOR, IndustryOR, OptiBench, and OptMATH remain unchanged. The corrections mainly address inconsistent or inaccurate released objective values, including suboptimal reference solutions, numerical precision issues, and incorrectly labeled solver outcomes. Unless explicitly stated otherwise, all results produced by our evaluation pipeline use the same benchmark versions.

To reduce potential train–test contamination, evaluation problems are excluded from the data used for memory construction. The evaluation sets themselves are never used to construct the memory repository.

Evaluation Protocol. Following prior work on LLM-based optimization modeling (Huang et al., 2025a; Jiang et al., 2025; Lu et al., 2025; Chen et al., 2026b), we evaluate a generated solution by executing its solver program and comparing the resulting objective value with the reference target. Let v denote the objective value obtained from the generated model and $v^\star$ denote the reference objective value. For instances with a finite reference solution, a prediction is considered correct when

$$\frac{|v - v^\star|}{|v^\star|} \leq 10^{-6}, \tag{10}$$

with an absolute tolerance of 10^{-6} used when the reference objective is sufficiently close to zero. A generated formulation must be executable by the optimization solver before its objective value is considered for scoring. For instances whose reference status is infeasible or unbounded, a prediction is counted as correct only when the solver returns the corresponding infeasibility or unboundedness status.

Following the evaluation convention adopted by recent optimization-modeling benchmarks (Lu et al., 2025; Chen et al., 2026b), we also account for benchmark instances in which variable-type specification is ambiguous.For an executable program that does not initially match the reference objective, we perform one variable-type consistency check by switching the relevant continuous and integer declarations and re-executing the formulation. The prediction is counted as correct if the resulting model matches the reference target under the same numerical tolerance. The same procedure is used for all ORCAS variants evaluated under this protocol.

For a benchmark $\mathcal{B}$, solving accuracy is computed as

$$\mathrm{Acc}(\mathcal{B}) = \frac{1}{|\mathcal{B}|} \sum_{p \in \mathcal{B}} \mathbb{I}\big[\mathrm{Correct}(p)\big]. \tag{11}$$

The overall score reported in the paper is the unweighted macro-average over the eight benchmarks,

$$\mathrm{Avg} = \frac{1}{8} \sum_{j=1}^{8} \mathrm{Acc}(\mathcal{B}_j), \tag{12}$$

so that benchmarks with different numbers of test instances contribute equally to the final average.

A.2 Baseline Details

We compare ORCAS with three groups of methods that represent different ways of providing or acquiring optimization modeling knowledge: general-purpose LLMs, prompt-based modeling methods, and models specifically fine-tuned for optimization modeling. The corresponding backbone model is reported explicitly for fine-tuned methods in the main results.

General LLMs We first evaluate representative general-purpose and reasoning-oriented LLMs without additional optimization-specific experience construction or parameter updates.

GPT-5 (OpenAI, 2026) is a general-purpose multimodal model with strong instruction-following and reasoning capabilities. We use it as a representative commercial LLM to evaluate the extent to which general language and reasoning capabilities transfer to optimization modeling.

DeepSeek-V4 (DeepSeek-AI, 2026) is a large general-purpose open-weight model with strong coding and reasoning capabilities. It serves both as a standalone baseline and as the common backbone for prompt-based methods whenever applicable, allowing us to isolate the contribution of different inference-time modeling strategies.

DeepSeek-R1 (DeepSeek-AI, 2025) is a reasoning model trained to perform extended multi-step reasoning through reinforcement learning. We include it to examine the effectiveness of reasoning-specialized models on optimization problems that require jointly constructing variables, objectives, constraints, and executable solver programs.

Qwen2.5-72B (Qwen Team, 2024) is an open-weight general-purpose large language model with strong instruction-following, coding, and mathematical reasoning capabilities. We adopt it as a standalone baseline and, where applicable, as a shared backbone for prompt-based methods, thereby isolating the contributions of different inference-time strategies.

Prompt-Based Methods Prompt-based methods improve optimization modeling without updating the underlying model parameters. They differ primarily in how they organize reasoning, search over candidate formulations, interact with solvers, or reuse previous modeling knowledge.

Chain-of-Experts (CoE) (Xiao et al., 2024) decomposes optimization modeling into multiple specialized roles that collaboratively perform problem understanding, mathematical formulation, code generation, debugging, and verification. This multi-agent decomposition is intended to reduce the burden of solving a complex modeling problem in a single generation.

OptiMUS (AhmadiTeshnizi et al., 2024) uses a modular LLM-agent workflow to construct mathematical models and solver programs, and iteratively revises them using execution and debugging feedback. Compared with direct prompting, it places greater emphasis on closing the loop between formulation, solver execution, and model correction.

AlphaOPT (Kong et al., 2025) improves optimization modeling through a self-improving experience library. It extracts reusable modeling insights from solver-verified problem-solving trajectories and iteratively refines the stored knowledge to guide subsequent problems.

OptiTree (Liu et al., 2025) organizes optimization modeling knowledge into a hierarchical tree of problem structures and modeling thoughts. For a new problem, related modeling knowledge is retrieved from the tree and used to guide hierarchical decomposition and formulation.

ReLoop (Lian et al., 2026) introduces structured modeling and behavioral verification. It checks whether the generated model matches expected problem behavior and then uses the detected mismatch to refine the formulation.

For controlled prompt-based comparisons, we use DeepSeek-V4 as the underlying model whenever the method supports backbone replacement.

Fine-Tuned Models We further compare against optimization-specific models that incorporate modeling knowledge into model parameters through supervised training, process supervision, or reinforcement learning.

ORLM (Huang et al., 2025a) trains a specialized operations research language model using OR-Instruct, a large-scale instruction dataset for optimization modeling. The model learns to translate natural-language problem descriptions into mathematical formulations and executable solver code through domain-specific supervised fine-tuning.

LLMOPT (Jiang et al., 2025) learns optimization modeling through a unified representation of problem definitions and large-scale instruction tuning. It aims to improve generalization across heterogeneous optimization problems by jointly learning structured problem understanding, mathematical formulation, and solver-code generation.

OptMATH (Lu et al., 2025) uses a scalable bidirectional data-synthesis framework to construct optimization modeling training examples. Its synthetic data pipeline connects optimization models with corresponding natural-language descriptions, enabling specialized models to learn from substantially enlarged solver-verifiable training corpora.

SIRL (Chen et al., 2026b) introduces solver-informed reinforcement learning for optimization modeling. Generated solver programs are executed and evaluated using solver-derived signals, allowing the policy to learn from verifiable outcomes rather than relying exclusively on static supervised demonstrations.

StepORLM (Zhou et al., 2026) introduces a step-wise training framework for optimization modeling. It integrates solver-verifiable data construction with solver-informed reinforcement learning, enabling the model to learn from executable outcomes and iterative refinement.

For previously released fine-tuned baselines, we retest the results associated with the corresponding backbone models. We explicitly retain the backbone name for each model because both the training method and underlying model capacity can substantially affect optimization modeling performance.

A.3 Implementation Details

This appendix documents the configuration under which the reported numbers were produced: the orchestration infrastructure, the sandboxed execution environment, the deterministic numerical gate, the read-only memory protocol used at evaluation time, and the controls that keep each ablation a single-variable comparison.

Backbone and decoding configuration. All four ORCAS roles are instantiated on the same backbone, DeepSeek-V4; no role uses a stronger or larger model than any other. Decoding parameters are held identical across roles and across every configuration in the paper: temperature 0, top-p 1.0, and a single sample per invocation. We use neither self-consistency, best-of-n sampling, nor answer voting, so each reported accuracy reflects one trajectory per problem rather than the best of several. All configurations within a table are run against the same API version, so cross-row differences are attributable to architecture rather than to provider-side model updates.

Orchestration and artifact persistence. The top-level scheduler is a deterministic program rather than an additional language-model agent, so that no part of the reported behaviour depends on an orchestrator's own reasoning. It dispatches each role in sequence, waits for that role's reply, extracts the declared artifact, and writes it to disk before the next role is invoked. Every artifact is indexed by problem identifier and correction cycle, so cycle k never overwrites cycle $k-1$; the complete trajectory $\mathcal{H}_t$ of Equation 7 is therefore recoverable after the fact, which is what makes the failure typology of Appendix B.10 auditable. A role is considered finished only when its reply has been stable for a fixed quiescence window and the expected artifact has been parsed successfully; a malformed or absent artifact triggers re-dispatch of the same role, up to three attempts, after which the problem is recorded as a failure rather than silently skipped. Inter-agent communication is restricted to these file-mediated artifacts: an agent receives the role-specific projection $\mathcal{I}_i^k$ of the task state and never the raw conversational context of another agent.

Before each evaluation problem, the conversational context of all four agents is explicitly cleared. This is a hard requirement for the validity of the reported accuracies: without it, a residual formulation from problem p would remain visible while solving problem $p+1$, and benchmark scores would depend on presentation order. Clearing the context is distinct from clearing memory; the read-only memory snapshot described below persists across problems by design, while conversational state does not survive a single problem boundary.

Sandboxed execution environment. All generated programs are executed inside a container image pinned for the duration of the experiments, launched with no network interface, a 512 MB memory cap, a single CPU, and a 60-second wall-clock limit. The image provides Python 3.11 with PuLP (CBC), OR-Tools CP-SAT, SciPy, CVXPY (Clarabel, ECOS, SCS), NetworkX, NumPy, and SymPy; no additional package may be installed at solve time, so the solver backend available to ORCAS is fixed and identical in every configuration. Host-side execution is prohibited, and the Coder is required to report the container's verbatim `stdout`, `stderr`, exit code, and runtime rather than a summary, which prevents fabricated execution traces from entering the trajectory. When the solver returns Infeasible or Unbounded, that status is propagated verbatim instead of being coerced to a numeric answer, since silently substituting a default value would make genuine infeasibility indistinguishable from a modeling error at the reviewing stage. Within a single cycle the Coder may repair at most three purely syntactic or API-level faults in its own program; data values and model structure may not be altered in such a repair, as any change at that level must instead surface as reviewer feedback.

Iterative result determination. The correction loop is capped at $K = 3$ cycles. If the cap is reached without an Approved verdict, or if the Reviewer fails to return a parseable verdict, the system falls back to the most recent answer produced by the Coder. Such fallback answers are scored exactly like any other prediction and are never discarded; this conservative convention ensures that the reported accuracies include unreviewed outputs rather than conditioning on successful verification. The sensitivity of accuracy to K is reported in Appendix B.6.

Frozen memory at evaluation time. ORCAS is a self-evolving system, which creates a methodological hazard for benchmark evaluation: if memory continued to be written while the evaluation

set was being solved, each instance would be solved by a slightly different system, and the reported accuracy would be neither reproducible nor interpretable. We therefore separate memory *construction* from memory *use* and freeze the memory between them.

Construction proceeds exactly as described in Appendix B.4: the 200 OR-Instruct problems are solved by the full system with memory writing enabled, and each agent consolidates memory into its own workspace. When this pass completes, the resulting per-role memory stores are snapshotted and the snapshot is the sole artifact carried into evaluation. Throughout the main experiments, each agent's workspace memory is mounted *read-only*: the consolidation step is disabled in the agent protocols, the memory directories are made non-writable at the filesystem level so that an attempted write fails rather than succeeding unnoticed, and the snapshot is verified by checksum before and after every benchmark run to confirm it was not modified. Formally, the update in the memory layer is suppressed, so that $\mathcal{M}_i^t = \mathcal{M}_i^{\text{snap}}$ for every agent i and every evaluation instance t, and the memory term in Equation 9 is a constant rather than a function of evaluation history.

Three properties follow, and each is a property we need in order to report a benchmark number at all. First, *order independence*: because no evaluation instance can alter the state visible to any other, permuting the evaluation set cannot change the score, and the per-benchmark accuracies of Table 1 are well defined. Second, *absence of test-time leakage*: a solved evaluation instance cannot deposit its own formulation into memory and thereby assist a structurally similar instance later in the same benchmark, which would otherwise inflate accuracy on benchmarks containing families of near-duplicate problems. Third, *comparability with the baselines*: prompt-based and fine-tuned baselines are static at inference time, so an ORCAS that kept learning during evaluation would not be compared against them on equal terms. The construction set is disjoint from every evaluation set, with overlapping instances removed before sampling, so the frozen snapshot contains no evaluation problem.

The same freezing discipline defines the *w/o Memory* condition. ORCAS(0) is not the full system with retrieval suppressed at query time; it is the identical system operating on an empty snapshot, with memory equally read-only. Consequently the ORCAS(0) versus ORCAS(200) contrast used both in the main results and as the memory row of Table 2 varies only the contents of $\mathcal{M}_i^{\text{snap}}$, while the architecture, the skills, the correction loop, the decoding parameters, and the evaluation protocol are untouched. The sizes reported in Table 8 are likewise produced by constructing a separate snapshot per budget and freezing each one before evaluation, rather than by truncating a single large store, so that each budget reflects memory that was actually consolidated under that budget. We note that this design measures the benefit of memory transferred from a disjoint construction corpus, and deliberately does not measure within-benchmark continual adaptation; the latter is a different and weaker claim, and we do not make it.

Single-variable control in the ablations. Every ablation in Section 4.3 is derived from the full ORCAS(200) configuration by changing exactly one factor and inheriting everything else unchanged. The invariants held fixed across all ablation rows are: the backbone and decoding parameters of Appendix A.3; the frozen memory snapshot of Appendix A.3; the container image, solver versions, and resource limits of Appendix A.3; the numerical gate and its tolerances; the correction budget $K = 3$; the evaluation subset and the order in which its instances are presented; and the scoring protocol of Appendix A.1, including the single variable-type consistency check, which is applied identically to every variant so that no configuration is scored more leniently than another. Each variant is run on the same four benchmarks as a fresh pass rather than being reconstructed from partial logs of the full system.

Removing a role requires care, because deleting an agent also deletes the work it performed, and a naive removal would confound the absence of the role with the absence of its output. We therefore reassign responsibilities rather than drop them. In *w/o Planner* the Modeler formulates directly from the raw problem statement and inherits the Planner's operating instructions relevant to data extraction, so the ablation removes explicit stepwise decomposition rather than removing data extraction itself. In *w/o Coder* the Modeler emits both the formulation and the solver program, executed in the same sandbox under the same limits, so the ablation removes the separation between formulation and implementation rather than removing execution. In *w/o Reviewer* the first generated program's result is taken as final; because no verdict exists, the correction loop cannot run, and this variant is the single case in which the effective correction budget differs from $K = 3$ by construction rather

than by choice. The routing ablations of Section 4.3 retain the Reviewer and the full budget and alter only the destination of feedback, so that routing policy is isolated from detection quality: the same faults are detected in all three routing variants, and only the agent receiving them changes.

Controlling context length in the skill and workspace ablations. Two ablations change how much text enters the model's context, which makes context length a potential confound, and we report the relevant token budgets so that the comparison can be judged directly. In the *50% skills* and *no skills* variants, reduced accuracy is accompanied by a *smaller* context, so the observed degradation cannot be attributed to context dilution; if anything, the shorter context favours the ablated variant, and the effect we report is therefore a lower bound on the contribution of the removed skills. The removed half is sampled once per role and held fixed across all four benchmarks and all problems, so a variant is not advantaged on a particular instance by a fortunate draw.

The *skills injected into the prompt* variant changes the delivery mechanism while keeping skill content byte-identical to the full configuration. Because the full library is concatenated unconditionally, this variant consumes more input tokens per invocation than the full system, which loads only a task-dependent subset. The comparison therefore reflects the joint effect of losing conditional selection and of operating under a longer context. We report this explicitly and note that the conclusion is robust in direction: the unconditional variant is both more expensive and less stable, so any advantage from longer context does not reverse the observed degradation. For reference, the full system's per-role fixed and dynamic token loads are given in Table 11; the unconditional variant replaces each role's task-dependent skill load with that role's entire library.

The workspace ablations of Section 4.3 hold total available knowledge constant while changing its partitioning. In *shared workspace, independent contexts*, the union of the four role documents and skill libraries is visible to all four agents, so no knowledge is removed relative to the full system and the manipulation is purely one of information scoping; the degradation therefore cannot be explained by missing content. In *shared workspace, shared context*, a single conversation performs all four roles in sequence, which additionally removes the context clearing between roles. As noted in the main text, the cell (isolated workspace, shared context) is not realizable on this platform because workspace isolation entails context isolation, so this ablation is nested rather than factorial, and the second row should be read as the compound effect of merging workspaces and merging contexts rather than as an independent context effect.

Token accounting. Token counts in Table 11 are measured with the backbone's own tokenizer on the exact strings dispatched to and returned from the API, and are averaged over solved problems. *Fixed role knowledge* comprises the role's operating documents and the skill material loaded for that problem; *problem and upstream artifacts* comprises the problem statement together with the upstream artifacts passed to that role. Reported figures are for a single cycle; a problem requiring k correction cycles incurs approximately the cost of the re-invoked roles for each additional cycle, which is why inner-loop routing to the Coder alone is materially cheaper than an outer-loop rollback through the Planner. Costs incurred by sandbox execution are not token costs and are excluded.

B Additional Experimental Results

B.1 Backbone Generalization

We first examine whether prompt-based ORCAS depends on a particular underlying LLM. In addition to the default DeepSeek-V4 setting, we build experiential memory and test modeling abilities using Qwen2.5-72 (Qwen Team, 2024), and DeepSeek-R1 (DeepSeek-AI, 2025).

As shown in Table 5, ORCAS consistently improves the performance of the corresponding backbone model across models with substantially different base capabilities. The improvement is not restricted to strong reasoning models, and the persistent gains across model families suggest that CLAWs-style modeling is not tied to any specific LLM.

Table 5: Backbone generalization with ORCAS. Accuracy (%) is reported on eight optimization modeling benchmarks. "ORCAS(0)" denotes the base framework without memory accumulation. "ORCAS(200)" denotes the framework that accumulates memory from 200 OR-Instruct (Huang et al., 2025a) problems.

Method	NL4OPT	MAMO Easy	MAMO Complex	NLP4LP	ComplexOR	IndustryOR	OptMATH	OptiBench
Qwen2.5-72B	74.6	94.8	41.4	88.2	50.0	38.0	15.1	54.6
Qwen2.5-72B + ORCAS(0)	94.1	95.0	70.3	95.1	61.1	48.0	32.6	65.8
Qwen2.5-72B + ORCAS(200)	**94.3**	**95.5**	**70.3**	**96.7**	**61.1**	**50.0**	**34.5**	**68.2**
DeepSeek-R1	76.1	89.7	48.6	90.4	55.6	45.0	34.9	67.2
DeepSeek-R1 + ORCAS(0)	95.0	90.7	75.5	96.5	66.7	55.0	50.0	76.3
DeepSeek-R1 + ORCAS(200)	**95.2**	**91.8**	**75.5**	**97.9**	**66.7**	**57.0**	**51.7**	**78.2**
DeepSeek-V4	78.4	93.6	52.3	92.1	55.6	45.0	39.8	78.6
DeepSeek-V4 + ORCAS(0)	96.3	94.3	78.2	97.6	66.7	55.0	54.4	85.8
DeepSeek-V4 + ORCAS(200)	**96.7**	**95.2**	**78.2**	**98.9**	**66.7**	**57.0**	**55.4**	**87.3**

Table 6: Matched-compute comparison. All methods use the same backbone (DeepSeek-V4) and operate under approximately 63K input tokens and 6 model calls per problem. Accuracy (%) is reported. Δ is relative to the stronger matched-compute baseline.

Configuration	NL4OPT	MAMO-EasyLP	IndustryOR	OptMATH	Mean
Best-of-N (N=5)	89.3	93.8	52.0	48.0	70.8
Iterative self-refinement	90.8	94.0	52.0	50.5	71.8
ORCAS (matched budget)	96.7	95.2	57.0	55.4	76.1
Δ vs. best baseline	+5.9	+1.2	+5.0	+4.9	+4.3

B.2 Matched-compute comparison.

To rule out the possibility that ORCAS's gains stem from a larger inference budget, we compare ORCAS(200) against two lightweight baselines under a matched token and call budget on the same four benchmarks and backbone (DeepSeek-V4). Both baselines are calibrated to approximately 63K input tokens and 6 API calls per problem statistically listed in the Appendix B.8: best-of-N sampling with $N = 5$, and iterative self-refinement with three rounds. As shown in Table 6, both baselines improve over single-pass prompting but remain below ORCAS on every benchmark. The margin is largest on IndustryOR and OptMATH, where formulation structure is the dominant bottleneck, and smallest on the relatively saturated MAMO-EasyLP. This pattern indicates that additional compute alone cannot fully account for ORCAS's advantage: additional samples or refinement rounds alone do not recover the gains brought by role separation, routed correction, and workspace isolation.

B.3 Paired Significance Test.

To assess whether the improvement of ORCAS(200) over ORCAS(0) is statistically significant rather than a fluctuation of the evaluation set, we perform a McNemar test on the paired per-instance outcomes across all 1,893 evaluation instances. Table 7 reports the contingency table. The two systems agree on 1,853 instances (1,596 both correct; 257 both incorrect) and disagree on 40: ORCAS(0) is correct while ORCAS(200) is incorrect on 11 instances, and ORCAS(200) is correct while ORCAS(0) is incorrect on 29 instances. The McNemar statistic with continuity correction is

Table 7: Paired per-instance outcomes of ORCAS(0) and ORCAS(200) across all 1,893 evaluation instances.

	ORCAS(200) correct	ORCAS(200) incorrect	Total
ORCAS(0) correct	1596	11	1607
ORCAS(0) incorrect	29	257	286
Total	1625	268	1893

$$\chi^2 = \frac{(|11 - 29| - 1)^2}{11 + 29} = 7.225, \quad p \approx 0.007.$$

The asymmetry is therefore statistically significant at the 1% level, confirming that the gain from memory accumulation is not attributable to random variation. Consistently across all three backbone models in Appendix B.1, ORCAS(200) outperforms ORCAS(0), further corroborating the statistical significance of the memory gain and confirming that it is not tied to a specific LLM.

B.4 Memory Construction Data Analysis

Memory Construction Data For prompt-based ORCAS, we construct the memory of agents from a relatively small collection of optimization modeling problems. Unless otherwise specified, we randomly sample 200 problems from OR-Instruct (Huang et al., 2025a) after removing instances that overlap with the evaluation sets. Regardless of whether the solution of ORCAS is correct or not, it is possible to generate relevant memories, including empirical memory, case-based memory, and counterexample memory.

The 200-problem setting is used throughout the main experiments. This relatively small construction budget allows us to evaluate whether the performance gain comes from memory accumulation rather than from building an increasingly large retrieval corpus. We further study the effect of the construction data source and size in Appendix B.4.

Data Efficiency We further examine whether ORCAS is sensitive to the amount and source of data used for memory construction. We first replace the default construction set with 200 problems from MAMO (Huang et al., 2025b). We then vary the number of OR-Instruct problems from 0 to 300 while keeping the remaining configuration unchanged. The 200-problem and 0-problem OR-Instruct settings correspond to the configuration used in the main experiments. All evaluation instances remain held out under every setting.

Table 8: Effect of the source and size of the data used for memory construction. Accuracy (%) is reported on representative benchmarks.

Construction Data	NL4Opt	MAMOEasy	IndustryOR	OptMATH
200 MAMOEasy	96.5	97.0	54.0	54.9
0 OR-Instruct	96.3	94.3	55.0	54.4
50 OR-Instruct	96.4	95.0	54.0	54.9
100 OR-Instruct	96.5	95.0	56.0	55.2
200 OR-Instruct	96.7	95.2	57.0	55.4
300 OR-Instruct	96.7	95.4	57.0	55.4

With the gradual construction of memory, it has a certain positive effect on modeling ability of ORCAS. Increasing the number of construction problems beyond 200 does not produce obvious improvements, suggesting that the quality of accumulated memory is more important than simply increasing data volume. The results obtained from an alternative construction source further show that the framework is not restricted to a particular collection, although closer distributional alignment can benefit individual benchmarks.

B.5 Comparison of Memory Forms

The memory consolidated by ORCAS during construction is not homogeneous. A completed solving trajectory leaves three qualitatively different kinds of residue, and the main experiments retain all of them. This appendix separates them to ask which form carries the transferable signal.

The three memory forms. *Empirical memory* records abstracted procedural regularities stated independently of the instance that produced them, such as the conditions under which a count variable must be declared integral, or the circumstances that require an explicit subtour-elimination family. It is prescriptive and generalizing. *Case-based memory* records complete solved instances, retaining a problem statement together with its accepted formulation, and is retrieved by similarity to the incoming problem for analogical reuse. *Counterexample memory* records failures: a rejected formulation, the diagnosis issued by the Reviewer, and the correction that resolved it. It is restrictive rather than prescriptive, constraining what the agent must not do without itself supplying a formulation.

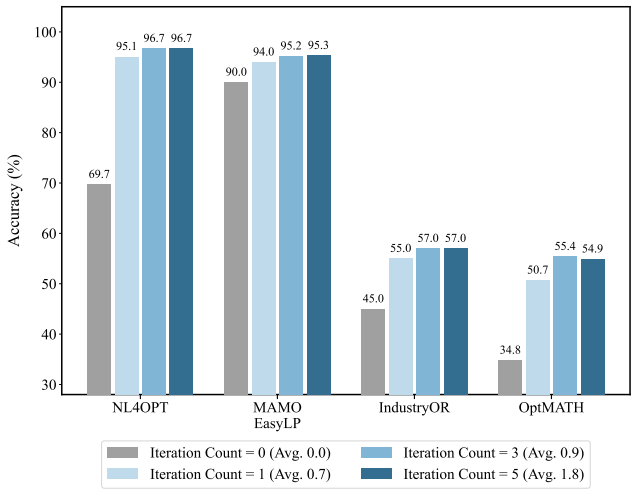


Figure 3: Sensitivity of solving accuracy (%) to the maximum number of repair iterations K.

Comparison of Three Memory Forms. To isolate the effect of memory form on optimization modeling, we compare three memory variants under a single-variable design, using ORCAS(0) as a reference to assess whether each form improves over no memory at all. Memory is constructed once on the same 200 OR-Instruct problems as ORCAS(200), and the three variants are derived by filtering this single consolidated snapshot so that each agent workspace retains only one memory form. All other factors are held fixed, and evaluation follows the same four benchmarks and scoring protocol as the remaining ablations.

Table 9: Comparison of memory forms. Each variant retains a single memory form, filtered from the same consolidated construction snapshot. Accuracy (%); Δ is the change in mean relative to ORCAS(0), the memory-free system.

Retained memory form	NL4OPT	MAMOEasyLP	IndustryOR	OptMATH	Mean	Mean Δ
Empirical only	96.4	95.0	54.0	55.2	75.2	+0.2
Case-based only	95.7	94.7	55.0	54.0	74.9	−0.1
Counterexample only	96.5	95.0	57.0	55.4	76.0	+1.0
ORCAS(0), no memory	96.3	94.3	55.0	54.4	75.0	—
ORCAS(200), all forms	**96.7**	**95.2**	**57.0**	**55.4**	**76.1**	+1.1

Table 9 shows that counterexample-only memory achieves the best single-form performance. The advantage of counterexample memory is concentrated on challenging benchmarks such as IndustryOR and OptMATH, whereas all variants perform comparably on simpler datasets. These results indicate that failure patterns transfer more effectively than episodic trajectories or distilled success cases.

B.6 Sensitivity analysis of maximum iteration times

We study the sensitivity of solving accuracy to the maximum number of repair iterations K, which controls how many times the Reviewer may trigger the correction loop ($K = 0$ disables repair entirely). We evaluate ORCAS(200) for $K \in \{0, 1, 3, 5\}$ on the four benchmarks, with all other settings held fixed. As shown in Figure 3, disabling repair yields a macro-average of only 59.88%, confirming that the correction loop is essential. A single repair round already raises the macro-average to 73.7%, with NL4OPT improving from 69.7% to 95.1% and OptMATH from 34.8% to 50.7%, indicating that most errors are corrected within one round. Increasing to $K = 3$ achieves the best macro-average of 76.1%, with best or tied-best accuracy on NL4OPT (96.7%), IndustryOR (57.0%), and OptMATH (55.4%), while consuming an average of 0.9 rounds. Further increasing to $K = 5$ does not yield consistent gains: the macro-average slightly decreases to 76.0% and

OptAMTH drops to 54.9%, despite the average rounds consumed rising to 1.8, suggesting that additional iterations may introduce over-correction or noise. Overall, $K = 3$ provides the best trade-off between repair sufficiency and iteration cost.

B.7 Reliability of the Reviewer Verdict

We evaluate Reviewer reliability on all eight benchmarks, comprising 1,893 problems of which 1,625 are solved correctly, by comparing its final verdict against final-answer correctness. As shown in Table 10, the Reviewer achieves 94.4% overall agreement, with a false approval rate of 9.7% and a false rejection rate of 5.0%; the APPROVED verdict has 98.3% precision and 95.0% recall over correct answers. The low false rejection rate indicates that correct answers are rarely sent back for unnecessary repair, while the higher false approval rate shows that residual errors concentrate in semantically consistent but problem-misaligned formulations, especially false-positive infeasibility and parameter-inference errors.

Table 10: Reviewer verdict versus final-answer correctness over 1,893 problems.

	Answer correct	Answer incorrect
Reviewer APPROVED	1,544 (TP)	26 (FP)
Reviewer CHANGES_NEEDED	81 (FN)	242 (TN)

B.8 Computational Cost

For ORCAS, the memory repository is constructed once and subsequently reused across evaluation problems. The construction process does not require gradient based parameter updates, only a complete solving process, mainly including analytical modeling steps, step-by-step modeling, coding and running, result review, iterative correction, and finally generating empirical memory. Therefore, the cost analysis of memory construction is the same as the entire process and does not require separate statistics.

Across all repair cycles triggered in the experiments under the maximum correction budget $K = 3$, 12% are routed back to the Planner (Plan Correction) and 88% are routed back to the Coder (Code Correction).

Token Cost Table 11 reports the token flow across the four stages of a complete ORCAS(200) solve. The total fixed cost for role-specific knowledge is 31.0K tokens, while the problem description and upstream artifacts contribute 14.2K tokens, yielding 39.8K effective input tokens. The total stage-visible output is only 1.8K tokens, indicating that the dominant cost lies in context loading rather than response generation. At the stage level, the Planner consumes 6.8K fixed tokens and 0.8K dynamic tokens, with 6.2K effective input and 0.6K visible output. The Modeler incurs the highest fixed cost (9.0K) and an effective input of 9.2K, producing 0.4K visible output. The Coder uses 6.9K fixed and 2.1K dynamic tokens, resulting in 9.0K effective input and 0.3K visible output. The Reviewer exhibits the largest dynamic cost (10.1K) and effective input (15.4K), as it must inspect the problem, the mathematical model, the generated code, and the solver result, while its visible output remains modest (0.5K).

Time Cost Each stage cost decomposes into three parts: generation latency, which scales with the stage's input and output tokens; for the Coder, sandbox execution; and orchestration overhead, the fixed delay the file-mediated relay adds while it polls for and confirms a stable reply.

Overall, compared to prompt-based single conversation modeling method, ORCAS does have a higher time cost, but it is still within an acceptable range and brings higher accuracy, as well as intermediate step files that can be reviewed and further fine tuned.

B.9 Quality and Interpretability of Intermediate Artifacts

Solving accuracy measures only the terminal answer and is silent on the quality of the representations that produced it. A formulation may yield the reference objective while leaving its coefficients

Table 11: Token flow across the four stages of a complete ORCAS solve.

Stage	Fixed Role Knowledge	Problem Upstream Artifacts	Input	Output	Primary Output
Planner	6.8K	0.8K	6.2K	0.6K	Workflow JSON
Modeler	9.0K	1.2K	9.2K	0.4K	Mathematical model
Coder	6.9K	2.1K	9.0K	0.3K	Python code and result
Reviewer	9.3K	10.1K	15.4K	0.5K	Verdict and feedback
Single forward pass	31.0K	14.2K	39.8K	1.8K	—
Average per problem	48.2K	25.4K	63.3K	2.6K	—

Table 12: Per-stage and end-to-end time cost of one ORCAS solve.

Stage	Agent calls	Sandbox times	latency (s)
Planner	1	–	12
Modeler	1	–	20
Coder	1	1	25
Reviewer	1	–	15
Forward pass	4	1	72
Plan Correction (once)	4	1	76
Code Correction (once)	2	1	42
End-to-end (per problem)	6.0	1.9	113

untraceable. Because ORCAS writes every intermediate representation to disk, these properties can be assessed directly. We therefore audit the mathematical model as an artifact in its own right, independently of whether the final answer was scored correct.

From each benchmark in NL4Opt, MAMO EasyLP, IndustryOR, and OptMATH, we randomly sample 50 instances that ORCAS(200) solved to completion, yielding 200 model artifacts in total. All sampled models are drawn from the frozen evaluation runs reported in the main experiments, so they reflect the same system configuration and decoding parameters. Each artifact is independently scored by five annotators, all operations-research researchers. Annotators are provided with the original problem statement and the extracted model artifact, but are **blind** to (i) whether the final answer was scored correct, (ii) the reference objective or solver status, and (iii) the identity of the system variant that produced the model.

Assessment rubric. We define seven dimensions for the model stage, each scored on a five-point scale, and apply them to sampled instances from four benchmarks spanning the difficulty range: NL4Opt, MAMO EasyLP, IndustryOR, and OptMATH. The dimensions are stated in Table 13. They are deliberately restricted to properties that can be checked by inspection against the problem statement and the upstream artifact, so that scoring does not require knowledge of the reference answer. For the model stage, these dimensions cover variable specification, objective correctness, constraint provenance, treatment of missing data, parameter documentation, assumption disclosure, and structural completeness of the assembled model.

Results. Table 14 reports mean scores by dimension and benchmark. The model-stage scores are uniformly high, with benchmark means ranging from 4.4 to 4.9. The strongest dimensions are variable specification (M1), constraint provenance (M3), and model completeness (M7), all scoring 5 on every benchmark, indicating that the core formulation is consistently well-structured and traceable. Notably, IndustryOR achieves a full score on assumption disclosure despite its complexity, suggesting that harder, more realistic problems may elicit more explicit assumption statements rather than weaker ones. Overall, the numerical pattern confirms that ORCAS(200) produces high-quality intermediate models, with residual weaknesses confined to documentation and data-gap handling rather than to the core mathematical formulation.

Table 13: Assessment rubric for intermediate models.

	Dimension	Criterion for a full score
M1	Variable specification	Every variable carries name, meaning, type, and bounds; integrality justified by physical indivisibility
M2	Objective correctness	Direction correct and evidenced by a verbatim quotation; every coefficient traceable
M3	Constraint provenance	Each constraint numbered and annotated with its source sentence; no constraint without textual basis
M4	Missing-data treatment	Gaps declared explicitly; inference method documented; infeasibility not asserted for recoverable data
M5	Parameter documentation	Parameter table gives value, unit, and source for every quantity
M6	Assumption disclosure	Assumptions section covers temporal structure, fixed costs, penalty semantics, and strict inequalities
M7	Model completeness	Assembled model contains variables, objective, constraints, parameters, and assumptions as a separable section

Table 14: Mean artifact quality by dimension and benchmark (1–5).

Stage	Dimension	NL4Opt	MAMOEasyLP	IndustryOR	OptMATH
Model	M1 Variable specification	5	5	5	5
	M2 Objective correctness	5	5	4	5
	M3 Constraint provenance	5	5	5	5
	M4 Missing-data treatment	5	5	3	4
	M5 Parameter documentation	5	4	5	3
	M6 Assumption disclosure	4	4	5	4
	M7 Model completeness	5	5	5	5
	Model mean	**4.9**	**4.7**	**4.6**	**4.4**

B.10 Distribution and Typology of Test-Time Failures

Answer-level scoring records whether an instance failed but not where or why. Because ORCAS preserves the complete artifact chain, each failure can be traced to the representation at which it originated. We audit all instances drawn from NL4Opt, MAMO EasyLP, IndustryOR, and OptMATH solved by ORCAS (200), separating execution failures from semantic ones and classifying every incorrect answer by root cause.

Execution and feasibility. All programs terminated with exit code zero. No residual accuracy loss is therefore attributable to non-executable code, unhandled exceptions, or solver timeouts. This indicates that the bottleneck of low accuracy in complex problems lies in the formulation stage, not in code execution.

We distinguish the *feasibility rate* – the proportion of approved answers that are numeric rather than INFEASIBLE or UNBOUNDED – from correctness. The two are independent: an infeasibility verdict is correct when the reference status is itself infeasible and incorrect otherwise. Table 15 reports both the rate and the provenance of the infeasibility verdicts.

Table 15: Feasibility rate and provenance of INFEASIBLE verdicts. A low rate is not necessarily a defect: on NL4Opt most such verdicts are correct.

Benchmark	Feasibility rate (%)	Provenance of INFEASIBLE verdicts
NL4Opt	91.8	85.7% correct (reference is infeasible); 14.3% false positive
MAMOEasyLP	100.0	none issued
IndustryOR	94.0	100% false positive (reference is numeric)
OptMATH	95.2	66.7% correct; 33.3% false positive
Overall	94.4	—

The provenance column is what makes the rate interpretable. NL4Opt has the lowest feasibility rate of the four benchmarks, yet most of its infeasibility verdicts are correct recognitions of genuinely infeasible instances; IndustryOR has a higher rate, but every one of its infeasibility verdicts is an error.

Failure taxonomy. Table 16 roughly classifies the observed failures into eight modes, with the share of audited instances exhibiting each. We assume that there is only one maximum problem for each instance.

Table 16: Failure modes as a share of incorrect instances.

	Failure mode	Share (%)
E1	False-positive infeasibility	41.3
E2	Error induced by inferred parameters	17.4
E3	Coefficient or magnitude error	13.0
E4	Omitted constraint or period	8.7
E5	Trivial zero solution	8.7
E6	Reversed objective direction	2.2
E7	Variable-type misjudgement	2.2
E8	Supply–demand structure misparsed	2.2

Grouping the modes by the stage at which they originate: interpretation of absent data accounts for 58.7% of instances (E1, E2), defects of formulation semantics for 30.4% (E3, E4, E5 in part), and answer extraction and variable typing for the remainder. No failure is executional. The distribution indicates the acceptance gate cannot detect a model that is internally consistent and misrepresents the problem, especially in parameter extraction. Further gains therefore depend on strengthening semantic audit, specifically, on calibrating the threshold at which absent data is treated as recoverable, which were present in the skill library but do not cover all issues.

C Case Studies

This appendix traces three problems from the main experiment through the complete artifact chain that ORCAS writes to disk. Case C.1 shows a problem approved on the first cycle. Case C.2 shows the outer rollback route, in which the Reviewer returns control to the Planner. Case C.3 shows the inner repair route, in which only the Coder is re-executed.

C.1 Case A: First-Cycle Approval on a Lot-Sizing MILP

Instance. OptMATH-Bench problem 114: a multi-item, multi-machine lot-sizing problem with 3 products, 2 machines, and 3 periods, combining setup and startup costs with inventory holding and backlogging. The instance carries 16-digit cost and demand coefficients, which makes faithful transcription the dominant failure risk.

Cycle-1 plan. The Planner classifies the problem and emits a seven-step workflow without writing any mathematics: (1) extract and verify input data; (2) define decision variables; (3) formulate the objective; (4) formulate inventory/backlog balance; (5) formulate machine capacity and one-product-per-machine constraints; (6) formulate startup logic; (7) select solver and verify feasibility.

Cycle-1 model. The Modeler executes the plan step by step and records the provenance of every coefficient. The declared variables are

$$y_{pmt},\, z_{pmt} \in \{0,1\}, \qquad x_{pmt},\, I_{pt},\, B_{pt} \;\geq 0,$$

where y_{pmt} indicates production of product p on machine m in period t, z_{pmt} indicates a startup, x_{pmt} is the production quantity, and I_{pt}, B_{pt} are inventory and backlog. The objective and constraints are

$$\min Z = \sum_{p,m,t} \big(143.7514410203047\, y_{pmt} + 65.60228476581376\, z_{pmt}\big) + \sum_{p,t} \big(h_p I_{pt} + b_p B_{pt}\big),$$

$$\text{s.t.} \begin{cases} I_{pt} - B_{pt} = I_{p,t-1} - B_{p,t-1} + \sum_m x_{pmt} - d_{pt} \\ I_{p,-1} = B_{p,-1} = 0 \\ -\mathrm{MC1}_m\, y_{pmt} + \mathrm{MC2}_m\, z_{pmt} + x_{pmt} \le 0 \\ \sum_p y_{pmt} \le 1 \\ z_{pmt} \ge y_{pmt} - y_{p,m,t-1}, \quad t \ge 1. \end{cases}$$

The model file states `data_completeness: COMPLETE`, records every coefficient at full precision alongside the quoted source phrase, and documents the Period-0 convention $y_{p,m,-1} = 0$ as assumption A5. The integrality decision is justified semantically rather than by keyword matching: production and startup decisions are yes/no and therefore binary, whereas quantities, inventory, and backlog are divisible and therefore continuous.

Cycle-1 code and result. The Coder emits a self-contained PuLP/CBC program in which equation C.1 appear as named constraint families `C1–C4`, and additionally prints a residual check for each of the nine balance equations. Execution in the sandbox returns `Solver status: Optimal` with a single production lot per product, balance residuals below 10^{-6}, and `Answer: 914.663638`.

Cycle-1 review. All four audit layers pass:

> All four layers pass. The lot-sizing MILP is fully data-complete (full precision) and faithfully implemented. Independent verification confirms all 9 inventory/backlog balance equations, all machine-capacity constraints ($x \le \mathrm{MC1} \cdot y - \mathrm{MC2} \cdot z$), one-product-per-machine, and startup-logic constraints hold, and the objective $Z = 3 \cdot 143.7514410203047 + 3 \cdot 65.60228476581376 + \text{holding} + \text{backlog} = 914.663638$ matches the reported Answer.

The verdict is `APPROVED` with `target_agent` empty, so no repair is triggered and the trajectory closes after one cycle with five output files.

C.2 Case B: Outer Rollback on a Permutation Flow Shop

Instance. OptMATH-Bench problem 73: a six-job, three-machine permutation flow shop. The objective sentence in the problem text refers to *the start time of the last job on the last machine plus its processing time* and instantiates it as $StartTime[5, 3]$, even though the machine index set is $\{0, 1, 2\}$. This off-by-one reference is the seed of two successive misjudgements, and the case therefore illustrates progressive repair across the full `MAX_FIX_CYCLES` budget.

Cycle 1: a false infeasibility from an uninstantiable term. The Modeler declares the data `PARTIAL` because $StartTime[5, 3]$ lies outside the machine range, and the executed program prints a data-completeness report instead of building a model:

> `Data completeness: PARTIAL`
> Machine index 3 in objective completion term `StartTime[5,3]` is OUT OF the valid machine range {0,1,2}; … No solver is invoked …
> `Answer: INFEASIBLE`

The Reviewer rejects this and routes the defect to `target_agent: planner-agent`, because the fault lies in problem interpretation rather than in code:

> False INFEASIBLE: … the problem text explicitly defines this term as the makespan: 'the start time of the last job on the last machine plus its processing time.' … the completion term can be faithfully modeled (completion pseudo-machine index 3, or a makespan variable $C_{\max}$), making the objective well-defined. The constraint set (permutation + precedence) is feasible, so INFEASIBLE is not established.

The accompanying `fix_instructions` prescribe the two admissible formulations (a completion pseudo-machine or an explicit $C_{\max}$ variable) and forbid emitting `INFEASIBLE` before the completion term has been modelled.

Cycle 2: the first defect is repaired, a second appears. The regenerated plan now models the completion term explicitly, and the `problem_summary` changes accordingly, from a generic makespan statement to *"modeled as `StartTime[5,3]` (completion term of the last job on the last machine) + the verbatim `JobSchedule[j][5]` coefficients* $(5, 6, 10, 7, 8, 8)$*"*. The objective is now well defined. The Planner, however, reads the model's phrase *"exact coefficients as detailed in the LP data"* as a reference to external data and declares the precedence-constraint coefficients missing, so the solver is again not invoked. The Reviewer identifies the new root cause and again routes to the Planner:

> False INFEASIBLE: ... the coefficients of the machine-precedence (C3) and job-precedence (C4) constraints are exactly the processing times $p[j][m]$, all of which are fully enumerated in the statement ... The phrase 'exact coefficients as detailed in the LP data' introduces no new numeric data — it only references how the fully-given instance is stored.

Cycle 3: approval. With both misjudgements removed, the Coder solves the complete MILP and reports the optimal permutation $(0, 2, 3, 4, 5, 1)$ over positions $0..5$ with $StartTime[5][3] = 63$ and `Answer: 69.000000`. The Reviewer confirms the value by back-substitution:

> All four layers pass. ... Independent verification confirms the permutation (pos: job $0, 2, 3, 4, 5, 1$) satisfies all C1/C2/C3/C3c/C4 constraints and yields objective $Z =$ coeff[job1] $+ StartTime[5][3] = 6 + 63 = 69$.

Files signature of the outer route. Because both repairs were routed to the Planner, the pipeline re-executed the full Planner → Modeler → Coder chain on each cycle. The directory therefore holds three complete artifact sets — three plans, three models, three programs, three results, and three feedback files. The two rejected cycles are preserved rather than overwritten, so the diagnosis that produced each revision remains auditable after the fact.

C.3 Case C: Inner Repair on a Capacitated Transportation LP

Instance. OptMATH-Bench problem 44: a seven-city capacitated transportation problem. The statement enumerates a 7×7 directed cost matrix, a 7×7 directed arc-capacity matrix, per-city incoming capacities, and net balances — 90 numeric values in prose form.

Cycle 1: a transcription failure disguised as a data gap. The cycle-1 model refers to the two matrices only as *"see problem text"* instead of reproducing their values. The generated program is consequently not an optimization model at all but a completeness validator that enumerates the 42 arcs it believes to be unspecified:

> `Data completeness status: PARTIAL`
> Arc cost matrix `c_ij` (42 directed arcs): GAP — referenced as 'see problem text' but numeric values NOT reproduced in the model file.
> ... `Answer: INFEASIBLE`

Diagnosis and routing. The Reviewer establishes that the data is in fact recoverable from the problem text, so the fault is an implementation defect rather than a modelling error, and sets `target_agent: coder-agent`:

> False INFEASIBLE. The problem statement fully enumerates the 7×7 directed transportation cost matrix (42 arc costs ...) and the 7×7 directed arc-capacity matrix (42 values) row-by-row ... The `model.md` and solve file referenced these matrices only as 'see problem text' without reproducing the numeric values, so c_{ij} and u_{ij} were incorrectly flagged as a GAP and the solver was not run. Structural-inference Check 4 (row-by-row prose parsing) PASSES ... so the INFEASIBLE is a transcription error, not a genuine data gap.

Cycle 2: Coder-only re-execution. The routed repair bypasses the Planner and the Modeler. The regenerated program transcribes both matrices in full,

```
c = [[0,1,2,3,2,3,3], [2,0,3,2,2,3,3], ..., [3,3,2,3,3,3,0]]
u = [[0,8,7,7,8,7,8], [8,0,9,8,7,7,8], ..., [9,8,8,7,9,8,0]]
```

builds the LP with $0 \le x_{ij} \le u_{ij}$, flow balance, and per-city inflow limits, and solves it:

```
x_21 = 1.000000; x_30 = 2.000000; x_51 = 1.000000
Minimum total transportation cost Z = 7.000000
```

The Reviewer approves the result and reports the supporting flow decomposition: city $2 \to 1$ ships 1 unit at cost 1, city $3 \to 0$ ships 2 units at cost 4, and city $5 \to 1$ ships 1 unit at cost 2, meeting both demands within all capacities.

Files signature of the inner route. In contrast to Case C.2, the directory holds one plan and one model but two programs and two results. The absence of Plan and Model from Cycle 2 is the on-disk signature of Coder-only repair: the upstream artifacts were still valid and were deliberately not regenerated.

D WORKSPACE AND SKILLS DETAILS

CLAWs provides the persistent workspace abstraction and file -based agent state. ORCAS builds upon thisabstraction by introducing an OR-specific role decomposition, typed modeling artifacts, routed double-loop correction, role-specialized procedural skills, and a verification pipeline tailored to mathematical optimization.

D.1 AGENT WORKSPACE IMPLEMENTATION

Section 3 defines each role agent as inhabiting a private workspace $\mathcal{W}_i = \langle \Pi_i, \Sigma_i, \mathcal{M}_i \rangle$. This appendix documents how that abstraction is realized on disk. We describe the directory layout shared by all four roles, the structure and function of each file class, the role-specific content of the four workspaces, and finally one complete protocol file as a worked example.

Each role is a directory. There is no agent code: an agent is a language model invoked with a working directory, and everything that distinguishes the different agents is text in that directory. Because the orchestrator clears the conversational context at every problem boundary, these files are the only state that survives from one problem to the next.

Two consequences follow. First, a role's behaviour is a reviewable artifact rather than an opaque configuration: changing how the Modeler decides variable integrality is an edit to one markdown file, scoped to that role, requiring no retraining and no change to the orchestrator. Second, the four workspaces are mutually invisible. No agent can read another's directory, achieving true independence.

Directory Layout All four workspaces share one layout, shown here for the Reviewer:

```
workspace-reviewer/
  AGENTS.md              protocol: mission, workflow, audit layers
  IDENTITY.md            role identity and mission statement
  SOUL.md                hard behavioural rules and output format
  USER.md                contract with the invoking system
  TOOLS.md               environment-specific notes
  MEMORY.md              consolidated persistent memory
  HEARTBEAT.md           autonomous task schedule
  memory/
    202x-xx-xx.md        experience in temporary storage
  skills/
    SKILLS.md            skill index and load-order routing
    semantic-audit/
      SKILL.md           trigger front matter + procedure body
      assets/
        audit_checklist.md
    anomaly-detection/
      SKILL.md
    infeasible-ruling/
      SKILL.md
```

The mapping to the formal triple is direct: Π_i is `AGENTS.md` together with `SOUL.md`, Σ_i is the `skills/` tree, and $\mathcal{M}_i$ is `MEMORY.md` together with the dated files under `memory/`.

Structure of Each File Class **`AGENTS.md` — the operating protocol.** The largest and most role-specific file. Every instance follows one skeleton: an ASCII pipeline diagram locating the

role among the other three; a session protocol listing the files to read on activation; an annotated skill inventory stating when each skill applies; upstream and downstream declarations naming what the role receives, reads, and delivers; a mission statement; typed input and output specifications; a numbered step-by-step procedure; a fix-cycle protocol governing behaviour on correction cycles; and completion criteria that make compliance checkable. The step-by-step section is marked *execute ALL steps, skip NONE* in all four roles.

`SOUL.md` — hard behavioural rules. Where `AGENTS.md` specifies the procedure, `SOUL.md` specifies the constraints, organized as numbered hard rules whose violation is declared to break the pipeline, followed by softer guidelines, explicit prohibitions, and an output template. The Coder's instance is representative: fifteen hard rules covering single-message replies, self-contained code, mandatory printing of every decision variable's value so the Reviewer can back-substitute, answer precision (six decimals, never two), saving the program before executing it, sandbox-only execution, and a data-completeness gate. Two are worth quoting for the failure modes they anticipate:

> **8. No fabrication.** NEVER invent or guess execution output. Report only what the container actually produces.
>
> **15. Data completeness gate.** ... If `"status": "PARTIAL"` → write a data-completeness validator as the FIRST thing in `main()`... Do NOT invent placeholder values for missing entries.

The second rule explains an artifact pattern that would otherwise look like a malfunction: programs that emit a completeness report and terminate without calling a solver are following this rule, not failing.

`IDENTITY.md` — role identity. A short field list: name, role, disposition, and mission. Its function is to fix the stance the agent takes toward its own output. The Reviewer's is the sharpest, declaring the role an *independent validator* and *the last line of defense against wrong answers*, and stating explicitly: *You are NOT a rubber-stamp; you are an adversarial verifier.* This matters because the default disposition of a language model asked to check work is agreement.

`USER.md` — contract with the invoking system. Names the audience as optimization experts and states what they judge. The Reviewer's instance makes the evaluation criterion explicit: the role is satisfactory if it evaluates modeling results objectively, and is penalized for failing to comply with `AGENTS.md` and `SOUL.md`. The four instances are near-identical in length, differing only in the role named.

`MEMORY.md` and `memory/`. The persistent memory of Section 3.1 is realized in two tiers. Dated files under `memory/` hold temporary memory distilled at the end of a session; `MEMORY.md` holds long term memory selectively stored based on temporary memory. It is the consolidated file that the session protocol of `AGENTS.md` instructs the agent to read on every activation.

Reserved and Unused Components For completeness we identify components present in the layout but not used in this deployment. `TOOLS.md` is a framework-provided template that is not consulted by any agent protocol in ORCAS; its content is therefore irrelevant to system behavior. `HEARTBEAT.md` is likewise unused, as the orchestrator invokes each agent on demand rather than through periodic activation. Neither file affects the reported results.

Example. We reproduce the structure of `workspace-reviewer/AGENTS.md`, the largest protocol file at 14,583 bytes, and `workspace-reviewer/SOUL.md`, as a concrete instance of the file class described in Appendix D.1. The listing below preserves the headings and the top-level ordering.

workspace-reviewer/AGENTS.md, abridged

```
# AGENTS.md - reviewer-agent

  [planner] --> [modeler] --> [coder] --> YOU (reviewer) --> decision
                                                    |
                              (fix: back to planner or coder)

You are the reviewer-agent, the quality gate and accuracy
```

```
enforcer of a closed-loop modeling system.

## Session Protocol
  1. Read SOUL.md      - hard constraints on your behavior
  2. Read IDENTITY.md  - your identity and core mission
  3. Read MEMORY.md    - important events and decisions
  4. Read USER.md      - user expectations of you

## Skills
  anomaly-detection/  - wrong-answer signatures  (always required)
  semantic-audit/     - Layer 1 and Layer 2 consistency checks
  infeasible-ruling/  - 7-node decision tree     (required when
                        the result contains INFEASIBLE)

## Upstream / Downstream
  Receives from: problem_id + cycle + original problem
  Reads:         plan, model, code, result  (4 files)
  Delivers:      verdict + diagnostic feedback JSON to pipeline

## Workflow
### Mission
### Inputs
### Outputs
### Step-by-Step Procedure (execute ALL steps, skip NONE)
  #### Phase 1: Read all four files
    {pid}_plan_c{k}.json    {pid}_model_c{k}.md
    {pid}_solve_c{k}.py     {pid}_result_c{k}.txt
  #### Phase 2: Four-layer audit - stop at the first failing
                 layer and issue CHANGES_NEEDED immediately
    Layer 1 - Problem vs Plan    (5 check groups)
    Layer 2 - Problem vs Model   (4 check groups)
    Layer 3 - Model vs Code      (2 check groups)
    Layer 4 - Result self-consistency (4 check groups)
  #### Phase 3: Issue verdict
  #### Phase 4: Reply Message
### Fix Cycle Protocol
### Completion Criteria
```

workspace-reviewer/SOUL.md, abridged

```
# SOUL.md - Reviewer Agent

## Hard Rules

## What You Do NOT Do

- You do NOT write code.          - You do NOT execute code.
- You do NOT communicate with the end user.
- You do NOT issue vague verdicts ("looks reasonable", "probably correct").

## Message Format

## Feedback JSON Schema

{
  "problem_id": "<id>",
  "cycle": "<review_cycle>",
  "Verdict": "APPROVED or CHANGES_NEEDED",
  "Answer": "<pure_number or empty string>",
  "target_agent": "planner-agent or coder-agent",
  "root_cause": "precise description of WHERE and WHY the error occurred,
                 naming the agent step",
  "issue": "exact description of what is wrong",
  "fix_instructions": "actionable instruction -- tell the agent to re-examine
                       a specific judgment",
  "diagnostic_questions": ["Question 1 ...", "Question 2 ..."],
  "modeler_fix_instruction": "if root cause is in modeler math derivation, the
                              specific correction; empty string otherwise"
}

**`diagnostic_questions` rules:**

**`modeler_fix_instruction` rules:**
```

The following excerpts are taken from the consolidated memory of the four workspaces after memory building process. Each agent writes only into its own store and in the vocabulary of its own role:

the Planner records classification and decomposition decisions, the Modeler records formulation patterns, the Coder records implementation and execution conventions, and the Reviewer records rulings and false-optimum signatures. The excerpts are abridged.

Planner memory

```
## Recurring classification and decomposition decisions

- Job-shop family: classify as MILP, not CP, when the problem asks only for a makespan
    value. Emit an explicit step instructing the Modeler to enumerate the shared-machine
    operation pairs BEFORE defining disjunctive constraints; when no machine-assignment
    table is present, the step must say so rather than assume the pairs can be derived.
- Coverage / tower-siting family: ILP. The data-extraction step must ask for the coverage
    matrix row by row. Phrases such as "similar coverage relationships as detailed in the
    data" indicate absent rows, not an inferable pattern -- mark PARTIAL in notes instead
    of instructing inference.
- Diet LPs: plain LP, continuous servings. Four steps suffice; adding a solver-selection
    step for these adds no value.
- Unit commitment: MILP with a period-0 initial-state trap. The constraint step must
    explicitly ask what the generator state is before the first period.

## Step-design lessons

- A step whose output is "the model" is too coarse; split into variables, objective,
    constraints. Reviewer Layer 1 rejects plans that jump from raw text to a full
    formulation.
- Every step's input must be the previous step's output. Introducing n without a prior
    step that extracts n was the cause of two Layer-1 rejections.
```

Modeler memory

```
## Recurring archetypes

- Job-shop MILP: continuous start times S(j,i) >= 0, continuous C_max, binary X per shared
    -machine pair. Disjunctive big-M in BOTH directions: S_A + p <= S_B + M(1-X_AB) and
    S_B + p <= S_A + MX_AB. Use the stated M (typically 100000); if none is stated, take M
     = sum of all processing times as a tight bound and document the choice.
- Coverage ILP: binary x_t (build), binary y_a (covered), maximize sum p_ay_a subject to
    budget. Linking must be sum_{t in COV(a)} x_t >= y_a. The reverse direction y_a <= ...
     is a silent error that still solves.
- Min-cost flow: verify supply equals demand before formulating. For unbalanced
    redistribution, net balance b_i = s_i - r_i and the constraint is outflow - inflow <=
    b_i; using >= makes the model infeasible whenever total excess is positive.
- Unit commitment: the initial state n(t,-1) = 0, so the period-0 startup linking
    constraint s(t,0) >= n(t,0) must be included. Omitting it understated the objective.

## Rules that repeatedly mattered

- Discrete counts (generators, employees, shortages) are integer; start times, flows,
    power output, and servings are continuous.
- Exactly-one assignment is an equality; at-most-one is an inequality.
- Structural absence is not inference. A stated pair count with no pair list, or coverage
    rows replaced by prose, is PARTIAL: deliver a validated template and name the missing
    parameter. Never fabricate the entries.
```

Coder memory

```
## Execution conventions

- Save the solve file before running; every invocation writes a fresh file, and each cycle
     gets its own name. Then execute in the sandbox and report the container's verbatim
    stdout, stderr, exit code, and runtime.
- Structure: main() plus an entry guard; suppress solver logs with PULP_CBC_CMD(msg=0);
    print every decision variable before the Answer line so the Reviewer can back-
    substitute.
- Precision: integers via int(result); continuous values to six decimals. Rounding to two
    decimals failed the tolerance check on the diet LPs.

## Data-completeness gate

- When the model reports PARTIAL, the validator runs first in main(): enumerate each
    missing parameter, print what is absent, print exactly "Answer: INFEASIBLE", and
    return without calling a solver. Placeholder values are never invented. This applied
```

```
    to the job-shop instances with no pair set, the coverage instances with missing rows,
    the staffing instances with no demand table, and one flow instance with no arc table.
- Infeasible and unbounded statuses are reported as such. Returning 0 silently would make
    a genuine data gap indistinguishable from a modeling error.

## Repair scope

- Within a cycle, only syntax, import, and solver-API faults may be repaired, at most
    three attempts, with the solve file rewritten each time. Data and model structure are
    never altered here; such changes must come back as Reviewer feedback. After three
    failures, report the error and mark the result so the Reviewer can see it.
```

Reviewer memory

```
## False-optimum signatures observed

- Unbounded helper variable inflated to erase a penalty: an attainment variable with only
    a lower bound produced Z = 0 while the assigned staff could not meet the requirement.
- Constraint applied over all indices without big-M deactivation, so unassigned staff
    forced the shortage up.
- Semantics that let the solver dodge a requirement: an empty project attains zero
    shortage, so the hard project is simply left unstaffed.
- Objective understated by an omitted initial-state term.

## Routing

Pure implementation defects go to the Coder. Wrong constraint form, missing bound, data-
    extraction gap, or reversed objective direction go to the Planner. A Modeler-specific
    instruction is attached only when the plan was correct and the mathematical derivation
     was not.
```

D.2 SKILLS

Skills are the mechanism by which ORCAS stores stable operations-research procedure outside the model parameters and outside the prompt. Each is a versioned document in the workspace of exactly one role, retrieved on demand according to the problem at hand. This appendix describes their on-disk layout, the schema every skill follows, the inventory held by each role, and one skill reproduced in full.

Directory structure. Every role workspace contains a `skills/` directory holding an index and one subdirectory per skill. The index, `SKILLS.md`, lists the available skills with a one-line description, a routing table mapping a task condition to the skill that must be read, and a summary of the role's capabilities; it is the only file the agent consults to decide what to open. Each skill subdirectory contains a single `SKILL.md` and, where the skill carries reusable material too long or too literal for prose, an `assets/` directory. The Coder's assets are runnable Python templates; the Modeler's and Reviewer's are lookup tables and checklists.

Skills directory layout, Modeler and Coder shown

```
workspace-modeler/
  skills/
    SKILLS.md                          index + routing table + capabilities
    workflow-planning/     SKILL.md
    data-completeness-audit/
                           SKILL.md
                           assets/audit_checklist.md
    constraint-semantics/  SKILL.md
                           assets/constraint_translation_table.md
    linear-optimization/   SKILL.md
    combinatorial-scheduling/
                           SKILL.md
    nonlinear-optimization/
                           SKILL.md
    goal-programming/      SKILL.md

workspace-coder/
  skills/
    SKILLS.md
```

```
    solver-discrete/        SKILL.md
                            assets/pulp_milp.py
                            assets/tsp_mtz.py
                            assets/dispersion_milp.py
                            assets/goal_programming.py
    solver-continuous/      SKILL.md
                            assets/slsqp_nlp.py
                            assets/cvxpy_socp.py
                            assets/networkx_mincostflow.py
                            assets/dp_knapsack.py
    code-execute-debug/     SKILL.md
```

Structure of a `SKILL.md`. All skills share one schema, which is what allows an agent to open a document at the section it needs rather than reading it end to end. A file opens with YAML front matter carrying `name` and `description`, the fields used for routing from the index. A *Metadata* table then records `skill_id`, `version`, `author`, `applies_to`, and `trigger`; versioning matters because a skill may be revised independently of every prompt that depends on it. *Purpose* states the error the skill prevents rather than the topic it covers. *When to Use* and *When NOT to Use* bound the regime of validity, and the second is the more important of the pair: an unbounded procedure is applied outside the conditions under which it was correct, which is the failure mode observed for empirical memory in Appendix B.5. The body is a sequence of numbered *Techniques*, each a judgment rule with its derivation, its trigger phrases, and a worked example. A *Trap Catalog* closes the file, one entry per known failure mode, giving the wrong form, the correct form, and the root cause. Traps carry namespaced identifiers (`LP-`, `CS-`, `SD-`, `SC-`, `GP-`, `NL-`, `ANOM-`), which lets a Reviewer diagnosis name a defect that lives in another agent's skill library without either workspace having to read the other. Documents are held to roughly 200–300 lines so that partial retrieval is meaningful; a single file large enough to require full loading would collapse on-demand retrieval into the unconditional prompt injection evaluated as an ablation in Section 4.3.

Skill inventory by role. Table 17 lists the fourteen skills and the condition under which each is read. *Always* denotes a skill read on every invocation of that role, *Step* one read when the plan reaches a particular step, *Type* one selected by the classified problem type, and *Conditional* one read only when a specific situation arises. Only four of the fourteen are unconditional, so the per-problem skill load is a small subset of the library, which is the source of the token gap between ORCAS and the prompt-injection variant discussed in Appendix A.3.

Example. We reproduce one technique and its associated trap verbatim from the Modeler's `constraint-semantics` skill. The defect it prevents yields a model that solves cleanly and returns a plausible number, so neither the solver nor execution feedback can expose it.

workspace-modeler/skills/constraint-semantics/SKILL.md, excerpt

```
---
name: constraint-semantics
description: Techniques for translating natural language constraint sentences
  into correct mathematical forms. Focused on judgment rules for direction
  ambiguity, OR/AND/IF-THEN encoding, cardinality, ratio, and absolute value
  -- with anti-patterns showing what goes wrong when the wrong form is used.
---

# Constraint Semantics -- Translation Techniques

## Metadata

| Field       | Value                                                          |
|-------------|----------------------------------------------------------------|
| skill_id    | constraint-semantics                                           |
| version     | 2.0.0                                                          |
| author      | workspace-modeler                                              |
| applies_to  | modeler-agent, constraint formulation step                      |
| trigger     | Read during the constraint formulation step for any problem    |

## Purpose

Translate ambiguous natural language into the mathematically correct constraint form. The
     wrong form produces a feasible model that silently violates the problem's intent --
```

Table 17: Skill inventory of the four role workspaces. Trigger indicates when a skill is retrieved: Always (every invocation), Step (a specific workflow step), Type (selected by problem type), Conditional (only in a named situation).

Role	Skill	Content	Trigger
Planner	problem-classification	Signal-word table, decision tree, hybrid resolution order, misclassification traps	Always
	workflow-design	Field specifications for each step, step-count guidance, per-type step sequences, 7 design traps	Always
Modeler	workflow-planning	How to execute each step type; data extraction, variable typing, objective, constraints, assembly	Always
	data-completeness-audit	Three-tier classification, 9 structural inference patterns, PARTIAL signal words	Step
	constraint-semantics	Direction rules, OR/AND/IF-THEN encoding, cardinality, ratio, absolute value	Step
	linear-optimization	Variable-type test, objective coefficients, LCM rule, big-M derivation, linking constraints	Type
	combinatorial-scheduling	Job-shop pair enumeration, MTZ depot exclusion, VRP load tracking, MCLP coverage, p-dispersion	Type
	nonlinear-optimization	NLP bounds and multi-start, SOCP cone recognition, arc enumeration, DP state design, analytical forms	Type
	goal-programming	Deviation pairs, goal equalities, priority weights, primary-quantity answer extraction	Type
Coder	code-execute-debug	Sandbox command, output-format rules, 7 debug categories with diagnosis and fix	Always
	solver-discrete	Library choice, integer declaration, answer extraction, status checks, big-M in code	Type
	solver-continuous	SciPy sign convention, free bounds, multi-start, CVXPY SOCP, NetworkX conventions, DP tabulation	Type
Reviewer	anomaly-detection	10 wrong-answer signatures with verification method per pattern	Always
	semantic-audit	Layer-1 and Layer-2 consistency checks, routing decisions, output formats	Always
	infeasible-ruling	Stub detection, 5-check structural inference, contradiction proofs, multi-cycle confirmation	Conditional

```
     either allowing solutions that should be forbidden (too loose) or forbidding solutions
      that should be allowed (too tight). Both types produce wrong answers without error
     messages.

## Technique 2 -- OR Constraint Encoding

### Why addition is always wrong for OR

"A or B" defines the feasible set as a union of two half-spaces. A sum A + B >= threshold
     creates a single half-space that is neither A nor B. It allows solutions where neither
      A nor B individually holds.

### Correct encoding: binary selector + two Big-M branches

Requirement: at least one of (expr_A >= threshold_A) or (expr_B >= threshold_B) must hold

Encoding:
  delta in {0,1}
  expr_A >= threshold_A - M(1-delta)    [branch A active when delta=1]
  expr_B >= threshold_B - Mdelta       [branch B active when delta=0]
  M = tight upper bound from problem data

### How to choose M for OR constraints

M must make the inactive branch non-binding. Rule: M >= max possible value of threshold_i
     minus the natural lower bound of expr_i.
```

```
### Trigger phrases for OR encoding

"or", "either ... or", "alternatively", "at least one of the following", "whichever", "one
      option is ... another option is"

### Example: "2 nurses OR 1 doctor must be on duty"

WRONG (sum):  2x_nurse + x_doctor >= 1   -- allows 0 nurses + 0.5 doctors
WRONG (AND):  x_nurse >= 2 AND x_doctor >= 1  -- requires BOTH, not OR

CORRECT:
  delta in {0,1}
  x_nurse  >= 2 - M(1-delta)    [2 nurses required when delta=1]
  x_doctor >= 1 - Mdelta       [1 doctor required when delta=0]
  M = max(2, 1) + safety margin from problem data
```

The corresponding hard rule appears in the Modeler's behavioural protocol ("any either/or or at-least-one-of constraint must use a binary selector with two big-M branches; never encode as a single additive inequality"), while the derivation, the choice of M, the trigger phrases, and the anti-pattern remain in the skill.

E PROMPTS

ORCAS dispatches six distinct messages: one per role on the forward pass, and one per repair target on a correction cycle. They are deliberately thin. Each message carries the task state a role needs and nothing else, because the role's standing behaviour lives in its workspace documents (Appendix D.1) and its domain procedure lives in its skills (Appendix D.2). Every message therefore opens by directing the agent to read its own protocol files rather than restating them, which is what keeps the dispatched text short while the effective instruction remains complete. Placeholders in braces are substituted by the orchestrator; paths shown as {EXP_DIR}/... are the artifact locations described in Appendix A.3.

Two properties of these prompts are load-bearing for the claims in the paper. First, no message contains the reference answer, the ground-truth objective, or any benchmark metadata; the only problem information supplied is the natural language statement. Second, each role receives only the upstream artifacts its own stage consumes, which is the concrete form of the role-specific projection $\mathcal{I}_i^k$ in Equation 9 – the Coder never sees the workflow plan, and the Planner never sees generated code.

E.1 FORWARD-PASS PROMPTS

The Planner receives the problem statement alone. The instruction not to solve the problem is explicit because the failure mode it prevents is a Planner that emits a formulation instead of a decomposition, which leaves the Modeler with nothing to execute and collapses the layer separation.

Planner, forward pass

```
You are the planner-agent. Read your SOUL.md, AGENTS.md then process the
following.

## problem_id
{pid}

## Problem Statement
{question}

## Instructions
- Output the **workflow plan** as a single fenced ```json block with key
  "workflow_steps".
- NEVER solve the problem. Only complete modeling plan and executable specific
  steps.
- Output in English + Markdown. One message only.
```

The Modeler receives the problem statement, the Planner's reply, and the path to the plan file. The reply and the file are both supplied on purpose: the inline reply is what the agent reads immediately,

while the path lets it reread the plan as written to disk, which is the artifact the Reviewer will later audit against. This message is the one place in the pipeline carrying an explicit error checklist. The five entries are not generic advice; they are the highest-frequency defects observed during construction, and the third is expanded in detail because variable-type misdeclaration was the single most common source of wrong answers.

Modeler, forward pass

```
You are the modeler-agent. Read your SOUL.md, AGENTS.md then process the
following.

## problem_id
{pid}

## Original Problem Statement
{question}

## Workflow Plan (from planner-agent)
{planner_reply}

## Workflow Plan File
{EXP_DIR}/plan/{pid}_plan_c{cycle}.json

## Common Failures to Avoid
1. Missing depot exclusion in MTZ constraints (TSP/VRP): u_i constraints must
   NOT apply to the depot node.
2. Confusing "maximize profit" with "minimize cost" when both appear: quote the
   exact objective phrase from the problem.
3. **Variable type -- INTEGER rule (most common error):** Declare a variable as
   `integer` (NOT `continuous`) whenever it counts discrete physical objects or
   events. Applies to: workers, nurses, volunteers, employees, drivers,
   animals, vehicles (trucks/vans/buses/cars/bikes), machines, bins, batches,
   boxes, shifts, trips, concerts, shows, sessions, doses -- and ANY variable
   where the meaning uses "number of X", "how many X", or "X per [period]" for
   a countable item. Physical indivisibility is the signal; the word "integer"
   does NOT need to appear in the problem text.
4. Omitting non-negativity constraints for allocation/quantity variables.
5. Missing linking constraints in big-M formulations (e.g., x_ij <= M * y_i).

## Instructions
- Read the workflow plan file. Follow EVERY step in workflow_steps exactly, in
  order.
- Output in English + Markdown. One message only. NEVER solve numerically.
```

The Coder receives the modeling spec and the model file path, and no part of the problem statement or the plan. This is intentional isolation: the Coder's task is faithful translation of the specification it was given, so supplying the original text would invite it to reinterpret the model rather than implement it, and a discrepancy between problem and model is the Reviewer's responsibility to detect. The save-before-execute ordering is stated in the prompt as well as in the workspace protocol because the orchestrator verifies the file's timestamp, and a program executed without a corresponding saved file cannot be audited afterwards.

Coder, forward pass

```
You are the coder-agent. Read your SOUL.md, AGENTS.md then process the
following.

## problem_id
{pid}

## Modeling Spec (from modeler-agent)
{modeler_reply}

## Model File
{EXP_DIR}/model/{pid}_model_c{cycle}.md

## Instructions
- **Save the code to disk FIRST:** `{EXP_DIR}/solve/{pid}_solve_c{cycle}.py`
  (required before Docker execution).
- Output in English + Markdown. One message only.
- Include verbatim execution output (exit_code, runtime, stdout, stderr) in your
  reply.
```

The Reviewer is the only role that receives the original problem together with every downstream artifact, since its audit is precisely a comparison across representations. It is given paths rather than inline content for all four artifacts, which forces it to read what was actually written to disk instead of auditing a copy relayed through the orchestrator.

Reviewer, every cycle

```
You are the reviewer-agent. Read your SOUL.md, AGENTS.md then review the
following.

## problem_id
{pid}

## review_cycle
{cycle}

## Original Problem
{question}

## File Paths for this cycle (read ALL four)
- Workflow plan:      {EXP_DIR}/plan/{pid}_plan_c{cycle}.json
- Mathematical model: {EXP_DIR}/model/{pid}_model_c{cycle}.md
- Solution code:      {EXP_DIR}/solve/{pid}_solve_c{cycle}.py
- Execution result:   {EXP_DIR}/result/{pid}_result_c{cycle}.txt


## Instructions
- Output in English + Markdown. One message only.
- Reply with "**Verdict: xx**" and include the feedback as a fenced ```json
  block in your reply.
```

E.2 Repair Prompts

A correction cycle dispatches one of two messages, selected by the routing rule of Section 3.2. Both announce the cycle index and the budget, so the agent knows how many attempts remain rather than repairing indefinitely.

The inner loop sends the Coder its own previous reply, the issue, and the fix instruction, together with the path to the *previous* cycle's model file. The model path points backwards deliberately: an inner-loop repair leaves the upstream artifacts untouched, so no new model exists for this cycle, and that absence is the on-disk signature of Coder-only repair discussed in the case studies. The new program is written under the current cycle index and must not overwrite its predecessor, which is what preserves the trajectory $\mathcal{H}_t$ for later inspection.

Coder, inner-loop repair

```
You are the coder-agent. This is fix cycle {cycle} of {MAX_FIX_CYCLES}.

## problem_id
{pid}

## Original Problem
{question}

## Previous Code (cycle {prev_cycle})
{prev_coder}

## Previous Model File (use as reference)
{EXP_DIR}/model/{pid}_model_c{prev_cycle}.md

## Issue
{issue}

## Fix Instructions
{fix_instructions}

## Instructions
- **Save updated code FIRST:** `{EXP_DIR}/solve/{pid}_solve_c{cycle}.py`
  (new file, do NOT overwrite previous)
- Output in English + Markdown. One message only.
- Include verbatim execution output in your reply.
```

The outer loop sends the Planner the diagnosed root cause, the repair instruction, and two conditional blocks. The *Diagnostic Questions* block appears whenever the Reviewer supplied questions, and it requires the Planner to answer each one at the top of its reply before revising anything. This ordering is the mechanism that forces a rereading of the original statement: a Planner permitted to rewrite its plan immediately tends to edit around the symptom, whereas answering a question such as which sentence fixes the objective direction obliges it to return to the source text. The *Modeler Fix Instruction* block appears only when the Reviewer judged the plan correct and the mathematical derivation faulty. Because the Modeler has no direct repair entry point, the correction is relayed through the Planner, which must embed it verbatim as a note on the relevant step; the Modeler then re-executes that step with the note in place.

Planner, outer-loop repair

```
You are the planner-agent. This is fix cycle {cycle} of {MAX_FIX_CYCLES}.

## problem_id
{pid}

## Original Problem
{question}

## Previous Planning Brief (cycle {prev_cycle})
{prev_planner}

## Root Cause Diagnosed by Reviewer
{root_cause}

## Fix Instructions from Reviewer
{fix_instructions}

## Diagnostic Questions (answer EACH before revising)      [conditional block]
Before updating the workflow plan and elements, answer every question below by
re-reading the original problem statement. Write your answers as a numbered
list at the top of your reply, then proceed with the revised plan.

{numbered diagnostic questions}

## Modeler Fix Instruction (relay to modeler in your updated workflow)
                                                          [conditional block]
The reviewer diagnosed a mathematical derivation error introduced by the
Modeler. Add the following correction as a note to the relevant workflow step
so the Modeler re-executes that step with the correction applied:

{modeler_fix_instruction}

## Instructions
- Output the revised **workflow plan** as a fenced ```json block
  (key: "workflow_steps").
- NEVER solve the problem. Only complete modeling plan and executable specific
  steps.
- Output in English + Markdown. One message only.
```