Agentic Repair of Gurobi Optimization Models via Tool Use: A Fractional-Factorial Study of Knowledge, Diagnostics, and Execution

Anonymous submission

Abstract

We present a modular agentic framework for automatic inspection and repair of Gurobi-based optimization code. The language model operates as an orchestrator over predefined tools and executes iterative reasoning through a closed loop of diagnosis, tool invocation, and verification. To measure the contribution of different functional tool capabilities, we apply a fractional-factorial experimental design using a newly constructed dataset of 26 Gurobi optimization tasks, evaluated across three random seeds, two model variants, and two reasoning budgets (12 vs. 24 steps). Our analysis shows that reasoning depth is the dominant factor governing repair accuracy: under shallow reasoning, knowledge-oriented retrieval and diagnostic tools yield substantial improvements in solver and validation performance, while under deeper reasoning their marginal benefit diminishes and execution feedback contributes little additional value.

Introduction

Large language models (LLMs) have demonstrated strong capabilities in code generation and editing, but recent studies show that text-only interactions remain fundamentally unreliable for program repair. LLMs often produce patches that appear correct in natural language yet fail at runtime, lack required imports, misuse APIs, or silently alter program logic (Bouzenia, Devanbu, and Pradel 2024). This limitation arises because debugging is inherently a stateful, multi-step process: a repair requires reading files, executing the program, inspecting errors, and iteratively refining the fix— activities that a pure text-in/text-out LLM cannot perform (Qiao 2025). Consequently, systems relying solely on prompting are prone to hallucinations, incorrect assumptions about program state, and inability to validate correctness (Qiao 2025; Kang 2025) .

These shortcomings have motivated a recent shift toward tool-augmented, agentic repair frameworks. Instead of emitting final code directly, the LLM operates as a controller over a set of tools, such as file readers, test runners, debuggers, or solvers. Agentic repair systems such as RepairAgent (Bouzenia, Devanbu, and Pradel 2024), InspectCoder(Kang 2025), and OR-LLM-Agent (Chen 2025) demonstrate that integrating program execution and structured observations substantially improves reliability. In these systems, the model issues structured function calls, observes the resulting tool output, and decides the next action in a closed loop. This execution-grounded workflow reduces hallucinations, enables fine-grained inspection, and allows the agent to verify each candidate fix through actual program behavior.

Our Work. We investigate how this agentic paradigm can be applied to automatically repair Gurobi-based optimization code. Unlike general Python, optimization scripts have strict structural requirements: variables and constraints must follow solver APIs, infeasibility must be correctly detected, and silent semantic errors can produce plausibly formatted but mathematically invalid models. Prior work shows that automatically repairing optimization code requires solver-feedback loops rather than static text generation (Chen 2025).

We therefore build an agent that uses OpenAI's function-calling protocol to orchestrate nine specialized tools, including file inspection, static analysis, program execution, and direct Gurobi model runs. The agent performs step-by-step reasoning, issues tool calls as JSON objects, observes structured feedback, and iteratively modifies only the relevant sections of the code. The process terminates either when the agent self-declares completion or when a predefined step limit is reached, mirroring realistic autonomous repair behavior rather than relying on our oracle unit tests. All tool calls, arguments, and outputs are logged as reproducible JSON traces, enabling deterministic replays and fine-grained analysis of repair trajectories. To investigate how different classes of tools and reasoning depth jointly affect repair performance, we construct a modular evaluation pipeline built on a benchmark of 26 Gurobi optimization tasks spanning linear, mixed-integer, combinatorial, and quadratic problem families. Because exhaustively enumerating all subsets of nine tools (2^{9-1} variants) is infeasible, we apply a fractional-factorial design to isolate the effects of three conceptual modules—Knowledge, Diagnostics, and Execution—under two reasoning budgets (12 and 24 steps). This approach allows us to quantitatively determine when tool augmentation meaningfully improves autonomous repair, and when additional reasoning alone is sufficient.

System Overview

Figure 1 shows the iterative repair loop. At each step, the agent receives the full conversation state (including previous

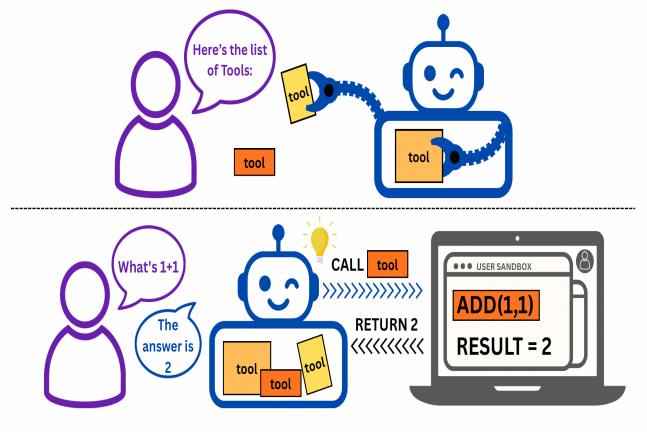

Figure 1: Agentic tool-use workflow: the model selects and executes tools, receives grounded feedback, and iteratively decides the next action.

tool outputs and available tools), selects one tool to invoke, executes it in a controlled sandbox environment, and incorporates the returned result into subsequent reasoning. The cycle repeats until either a completion signal is produced or the step limit is reached.

1. **Diagnose:** Read the full dialogue, tool call history.
2. **Act:** Select and call a tool from the enabled set.
3. **Execute:** Run the tool inside the sandbox.
4. **Reflect:** New reasoning based on the tool output.

Dataset and Setup

We constructed a benchmark of 26 Gurobi optimization problems spanning a range of modeling complexity and practical relevance. Thirteen problems were adapted directly from the official Gurobi documentation, such as workforce scheduling, production planning, and solution pool. The remaining thirteen were independently authored to model small-scale, interpretable applications (e.g., diet planning, regional budgeting, resource allocation) that illustrate general optimization principles in a controlled setting and cover representative ILP, LP, and MIP formulations.

Each problem is implemented as an independent, self-contained Python module defining:

- `solver_core()` — formulates and solves the optimization model,
- `validate_input()` — performs structured parameter checking, and
- a descriptive docstring summarizing the model objectives and constraints for the self-authored problems.

After assembling the 26 solver modules, we authored a suite of 10 unit tests per case using Python's `unittest` framework, yielding a total of 260 executable tests. Each suite evaluates both solver correctness and input validation:

- **Solver tests** include 138 feasible-case evaluations of optimal objective values and 14 infeasibility checks verifying correct detection of unsatisfiable models.
- **Validation tests** comprise 108 cases verifying that `validate_input()` differentiates valid parameter sets from malformed or inconsistent ones.

This dual testing framework provides comprehensive coverage of both modeling behavior and parameter validation, enabling precise, quantitative measurement of autonomous repair performance.

Experimental Setup

Bug Injection and Dataset Construction

Let $\{G^{(1)}_{\text{oracle}}, \ldots, G^{(26)}_{\text{oracle}}\}$ denote the 26 ground-truth optimization solvers (one per benchmark task). From each oracle implementation $G^{(i)}_{\text{oracle}}$, we generate three independent bug-injected variants via an LLM mutation procedure:

$$G^{(i)}_{\text{oracle}} \longrightarrow \{G^{(i)}_1, G^{(i)}_2, G^{(i)}_3\}, \qquad i = 1, \ldots, 26.$$

Mutations modify the `solver_core()` function to introduce realistic modeling defects (e.g., flipped objective orientation, altered constraint coefficients, corrupted index logic). All injected variants remain syntactically valid and executable, ensuring that evaluation reflects semantic rather than syntactic failure modes. The accompanying `validate_input()` routines are intentionally left blank, requiring the agent to synthesize or repair parameter-checking logic during debugging.

Repair Procedure

For notation, we view each corrupted benchmark variant as the set of 26 problem instances:

$$G_k = \{G^{(1)}_k, G^{(2)}_k, \ldots, G^{(26)}_k\}, \qquad k \in \{1, 2, 3\},$$

where each $G^{(i)}_k$ is a corrupted version of the i-th optimization task. Repairs are performed under experimental conditions determined by binary tool availability and reasoning-step budget. Each tool configuration is represented as

$$X = (K, D, E), \qquad K, D, E \in \{0, 1\},$$

corresponding to Knowledge, Diagnostics, and Execution tool modules. Two reasoning horizons are tested: $t \in \{12, 24\}$ where $t = 12$ uses GPT-5-nano and $t = 24$ uses GPT-5-mini as orchestrator.

For each corrupted benchmark G_k, configuration X, and reasoning budget t, the agent produces a repaired benchmark:

$$G^{X}_{k,\text{fixed}}\big|_t = \{\, G^{(1),X}_{k,\text{fixed}}\big|_t, G^{(2),X}_{k,\text{fixed}}\big|_t, \ldots, G^{(26),X}_{k,\text{fixed}}\big|_t \,\}.$$

All repaired programs are evaluated using our 260-unit-test harness, ensuring consistent quantitative comparison across reasoning budgets and tool configurations.

Aggregated Stability Across Step Budgets

To assess robustness under perturbation, performance is averaged across the three independently corrupted benchmarks $\{G_1, G_2, G_3\}$. We compare shallow and deep reasoning by aggregating repaired benchmarks across all tool configurations:

$$\mathcal{G}_{12} = \bigcup_X \bigcup_{k=1}^{3} G^X_{k,\text{fixed}}\big|_{12}$$

$$\mathcal{G}_{24} = \bigcup_X \bigcup_{k=1}^{3} G^X_{k,\text{fixed}}\big|_{24}$$

Table 1 reports solver and validation results over $\mathcal{G}_{12}$ and $\mathcal{G}_{24}$.

Metric	12 Steps	24 Steps	Δ (pp)	Change (%)
Solver Success (Feasible)	93.79	112.54	+18.75	+19.99
Solver Errored (Feasible)	33.33	16.88	−16.46	−49.37
Solver Failed (Feasible)	10.88	8.58	−2.30	−21.14
Solver Success (Infeasible)	7.71	5.92	−1.79	−23.22
Solver Errored (Infeasible)	5.54	6.96	+1.42	+25.63
Solver Failed (Infeasible)	0.75	1.13	+0.38	+50.67
Validation Pass	83.00	90.92	+7.92	+9.54
Validation Error	15.33	7.08	−8.25	−53.82
Validation Failed	9.67	10.00	+0.33	+3.41

Table 1: Mean **test counts** aggregated over three independently corrupted benchmark variants. Counts reflect the number of unit tests passed or errored across all tool configurations for each reasoning-budget condition. Solver metrics are split into feasible (138 tests per variant) and infeasible (14 tests per variant) categories; validation metrics reflect 108 structured input-check tests.

Interpretation. Extending the reasoning horizon from 12 to 24 steps substantially improves the agent's ability to repair feasible optimization behavior. On feasible solver tests, successful outcomes increase (93.79 → 112.54), while runtime errors nearly halve (33.33 → 16.88) and outright failures also drop (10.88 → 8.58). This suggests that additional reasoning cycles allow the agent to iteratively refine core model semantics (objective direction, constraint structure, index logic), and once a coherent Gurobi formulation is found, it generalizes smoothly across many input configurations.

In contrast, infeasible-case behavior remains brittle: successes decrease slightly (7.71 → 5.92), and both errors and failures grow. Unlike feasible instances, infeasible models do not provide a clear numerical trajectory to "pull" the agent toward the correct formulation; detecting infeasibility is a symbolic property of the constraint system, and small modeling changes can flip the outcome.

Validation exhibits only modest gains (83.00 → 90.92), reflecting a fundamental limitation of our setup: the agent never sees unit-test feedback while repairing. Fixed code is evaluated against the 260 tests only after the reasoning loop terminates, so the model must design input checks purely from conceptual reasoning and prompt instructions, without any hint about which edge cases it is currently missing. This is analogous to a student taking an exam without interim grading: they may capture the main patterns but still miss rare or finely tuned corner cases, leading to weaker improvements compared to solver repair.

Overall, deeper iterative reasoning reliably enhances semantic solver repair on feasible instances, while infeasibility handling and validation remain bottlenecked by the lack of runtime feedback and the discrete, combinatorial structure of error conditions.

Fractional Factorial Analysis of Tool Groups

To understand how different tool capabilities influence autonomous repair performance, we analyze the effects of the Knowledge (K), Diagnostics (D), and Execution (E) tool modules. Among the 9 total tools in our framework, three foundational utilities (`read_file`, `edit_file`, `write_file`) are always enabled, since disabling them would prevent any meaningful repair. These are excluded from factorial manipulation, leaving nine optional tools grouped into three conceptual modules.

Instead run $2^6 = 64$ full experiments, we apply a standard 2^3 fractional factorial design over {K,D,E}, yielding eight configurations per mutation instance. Values `0` and `1` indicate the absence or presence of each module. We first report raw performance under shallow and deep reasoning budgets, then compute main effects.

Raw Results Under 12-Step Reasoning

K	D	E	Solver Success (%)	Validation Pass (%)
0	0	0	62.08	76.23
0	0	1	65.46	76.85
0	1	0	71.26	79.01
0	1	1	63.77	83.46
1	0	0	68.36	75.62
1	0	1	74.64	73.77
1	1	0	68.84	79.01
1	1	1	69.32	73.15

Table 2: Mean solver and validation pass rates for all eight K–D–E configurations under the 12-step reasoning limit (averaged over three mutation variants).

Raw Results Under 24-Step Reasoning

K	D	E	Solver Success (%)	Validation Pass (%)
0	0	0	82.37	84.88
0	0	1	78.74	83.64
0	1	0	81.40	85.49
0	1	1	79.23	82.41
1	0	0	82.25	86.73
1	0	1	82.61	84.26
1	1	0	81.64	81.79
1	1	1	83.57	84.26

Table 3: Mean solver and validation pass rates under the 24-step reasoning limit, showing convergence and reduced dependence on external tool modules.

Main Effects Under Limited Reasoning (12 Steps) Under the 12-step setting, Table 2 shows a clear and immediate benefit from enabling any single tool group. Starting from the baseline configuration $(K = 0, D = 0, E = 0)$, turning on either Knowledge, Diagnostics, or Execution alone leads to a noticeable improvement in solver success. This indicates that, with shallow reasoning depth, the agent relies heavily on any additional structural support provided by the tools.

To quantify these contributions, we compute standard main effects by contrasting average performance between configurations where a factor is enabled versus disabled:

$$\text{Effect}(K) = \mathbb{E}[Y \mid K = 1] - \mathbb{E}[Y \mid K = 0],$$

and analogously for D and E, where Y denotes either the solver success rate or the validation-pass rate.

Factor	Solver (pp)	Validation (pp)
K (Knowledge)	+4.65	-3.50
D (Diagnostics)	+0.66	+3.04
E (Execution)	+0.66	-0.66

Table 4: Main effects of the Knowledge (K), Diagnostics (D), and Execution (E) factors under the 12-step setting, computed from the configuration means in Table 2. Positive values indicate that enabling a factor increases the corresponding pass rate (in percentage points) on average.

The 12-step results yield following factor-level conslusions:

- **Knowledge tools (K)** provide the strongest gains in solver correctness. This aligns with our setup: the knowledge module includes both online search and a local RAG index built from the Gurobi API documentation. Under shallow reasoning, the agent uses these retrieval cues (e.g., valid `Model.addConstr` patterns, attribute semantics, or solver parameter usage) to reconstruct missing logic that it cannot derive through reasoning alone.
- **Diagnostic effects on validation.** The Diagnostics factor shows a small and inconsistent effect on validation performance. At present, we do not have a clear explanation for this interaction. Our diagnostic tools focus on static structure and model introspection rather than input semantics, so there is no direct reason to expect improvement in validation behavior. We therefore treat this as an empirical observation rather than a causal conclusion. Understanding why Diagnostics exhibits minor influence on validation accuracy is an open question and an interesting direction for future analysis.
- **Execution tools (E)** do not help under shallow reasoning. Execution only shows results for a single test instance. If the agent cannot interpret those numeric outputs—or reason about why the model failed—it gains no advantage. For example, running the regional-investment model yields values and a solution vector, but a 12-step agent cannot infer from this alone whether constraints were missing, bounds mis-set, or the objective mis-specified. Thus, execution signals remain low-value until the model is capable of deeper multi-step reasoning.

Main Effects Under Deep Reasoning (24 Steps) We compute main effects for the 24-step setting using the same contrast-based method introduced in the 12-step analysis:

$$\text{Effect}(K) = \mathbb{E}[Y \mid K = 1] - \mathbb{E}[Y \mid K = 0],$$

with Y denoting solver or validation performance. The eight configurations in Table 3 provide the necessary averages for each factor level (0 vs. 1), from which the effects in Table 5 are derived.

With 24 reasoning steps, the agent becomes substantially more self-sufficient, and the marginal value of these tools decreases.

Factor	Solver (pp)	Validation (pp)
K (Knowledge)	+3.20	+3.03
D (Diagnostics)	+0.70	-0.40
E (Execution)	+0.10	+0.10

Table 5: Main effects of the Knowledge (K), Diagnostics (D), and Execution (E) factors under the 24-step setting, computed from the configuration means in Table 3. Tool contributions are substantially smaller than in the 12-step case.

The 24-step results reveal several clear trends:

- **Knowledge tools (K) remain mildly beneficial.** Even though deep reasoning allows the model to reconstruct much of the solver logic internally, the knowledge module supplies *ground-truth API corpus* from our curated Gurobi documentation RAG. Because the retrieved information is highly reliable (and domain-specific), it still nudges the agent toward correct modeling choices in both solver and validation outcomes.
- **Diagnostics (D) provides almost no additional value.** The small and slightly negative validation effect suggests that with ample reasoning steps, the model already performs sufficient internal self-checking. When the model can reason through its own code, external structural inspection becomes redundant and may even introduce mild

noise by diverting attention to secondary modeling details.

- **Execution tools (E) have near-zero marginal effect.** A single execution trace offers limited insight when the model already possesses enough reasoning depth to anticipate runtime behavior. Execution feedback only reports *what happened*, not *why it happened*; under deep reasoning, the agent can infer correctness directly from the symbolic structure of the code, making runtime feedback largely unnecessary.

Summary and Discussion

The fractional-factorial study reveals a coherent picture of how tool capabilities interact with the model's reasoning depth. While individual modules can provide meaningful benefits under constrained settings, deeper reasoning fundamentally changes the dynamics of repair.

Summary of Findings

Across both reasoning depths, a clear pattern emerges:

- Reasoning depth is the dominant driver of repair accuracy. Increasing the agent's step budget from 12 to 24 consistently improves both solver and validation performance, even with minimal tool support.
- Under shallow reasoning, Knowledge and Diagnostics tools provide the strongest gains. Knowledge tools supply reliable API semantics through local RAG and online search, while Diagnostics contributes structural guidance that the agent cannot infer with limited internal reasoning.
- Under deep reasoning, the agent effectively internalizes the repair process. External tools become supplementary rather than essential, and their marginal effects shrink to near zero.

This analysis provides a principled quantitative understanding of when tool augmentation matters and why. The benefits of external modules are not uniform—they depend critically on the model's reasoning budget.

Limitations

Execution feedback. Unlike traditional Python debugging—where a failing run typically reveals a stack trace or highlights the exact source of an error—Gurobi model execution returns only coarse solver signals (e.g., `OPTIMAL`, `INFEASIBLE`, objective values). A single run rarely indicates *which* constraint, variable, or structural modeling choice is responsible for failure. Because many modeling errors produce feasible yet semantically incorrect models, the Execution (E) module provides limited actionable guidance. This property is inherent to mathematical programming and explains why execution tools show minimal marginal effect in our fractional-factorial study.

Restricted knowledge corpus. The Knowledge (K) module draws from a small, API-focused RAG corpus containing roughly 300 entries (e.g., `tuplelist.select`, variable attributes, lazy updates). While this information is reliable, it does not include higher-level modeling patterns, canonical formulations, or examples of structural constraints used in practice. As a result, knowledge tools improve API usage but cannot supply broader modeling intuition, limiting their impact once the agent has sufficient reasoning depth.

Prompt sensitivity across tool configurations. Because each K–D–E configuration enables a different subset of tools, the agent receives slightly different system prompts for each setting. In particular, the Execution (E) module includes a scratchpad tool that requires a specialized instruction block encouraging the model to generate and run unit tests. This introduces mild heterogeneity across prompts. Although the differences are small and the core agent instructions remain identical, it is possible that prompt-level variation contributes to performance differences in addition to the tools themselves. Our study does not isolate these effects, and fully disentangling prompt sensitivity from tool contribution remains an open question.

Future Work

These limitations suggest several promising directions for strengthening tool-driven repair of optimization code:

- **Prompt-robust evaluation.** Because different K–D–E configurations enable different tools, each setting requires a slightly modified system prompt (e.g., the scratchpad tool necessitates explicit instructions on generating and running unit tests). Although the variations are small, prompt differences may still influence model behavior. Future work should develop prompts that adapt automatically to available tools or design a prompt architecture that remains invariant across tool subsets, ensuring that measured effects stem purely from functional tool capability rather than prompt-level confounders.
- **Richer diagnostic feedback.** Modern solvers expose powerful debugging primitives—such as conflict refinement, IIS (irreducible inconsistent subsystem) extraction, and constraint relaxation analysis—that pinpoint the exact components responsible for infeasibility or inconsistency. Integrating these into the Diagnostics module could transform coarse solver status codes into actionable, constraint-level repair signals.
- **Expanded modeling-oriented RAG.** Our current RAG corpus focuses primarily on Gurobi API semantics (roughly 300 entries). Expanding this corpus to include canonical modeling patterns (e.g., facility location, assignment, cutting stock), best-practice formulations, and domain-specific templates would enable the agent to perform higher-level semantic corrections, not just API-level fixes. Such modeling-aware retrieval may be especially beneficial under shallow reasoning limits.

Overall, these directions point toward more capable, diagnostic, and model-aware agentic repair systems—ones that exploit external tools when they provide genuine value and rely on internal reasoning when they do not. We plan to release the full benchmark suite, bug-injection scripts, and repair traces to support further research in agentic optimization debugging.

References

Bouzenia, I.; Devanbu, P.; and Pradel, M. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *arXiv preprint arXiv:2403.17134*.

Chen, e. a. 2025. OR-LLM-Agent: Automating Modeling and Solving of Optimization Problems with Large Language Models. *arXiv preprint 2503.10009v1*.

Kang, e. a. 2025. InspectCoder: Dynamic Analysis-Enabled Self-Repair via LLM-Debugger Collaboration. *arXiv preprint 2510.18327*.

Qiao, e. a. 2025. RepairAgent: An Autonomous LLM-Based Agent for Program Repair. In *ICSE*.