TREASURYOPT: BENCHMARKING LLMS ON GENERATING RELIABLE OPTIMIZATION PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly explored for mathematical optimization, yet existing evaluations rarely test whether they can construct reliable optimization programs that remain robust across unseen instances. We introduce TreasuryOpt, a benchmark evaluating LLMs on generating reliable optimization programs grounded in enterprise treasury operations, comprising four scenarios, 28 modeling cases, and 596 hidden test instances. For each modeling case, a system must construct a fixed program from its textual requirements, which is then evaluated across unseen instances without further model intervention. Our solver-verified evaluation protocol then extracts the returned decision variables and independently recalculates constraint satisfaction and objective optimality against solver-certified reference bounds. Extensive experiments across six frontier LLMs—under standalone and agentic settings across solver-hidden and solver-aware conditions—reveal a persistent *feasibility–optimality gap*: systems face considerable difficulty satisfying domain constraints, with success rates collapsing further when requiring solver-certified optimality. Furthermore, while interactive coding harnesses enhance solution quality via iterative self-debugging, they incur orders of magnitude higher generation costs with only limited performance gains. These results demonstrate that reliable optimization with LLMs requires progressing beyond surface-level executability toward mathematically sound and generalizable program generation.

1 INTRODUCTION

Large language models (LLMs) have achieved remarkable progress in automated code generation, driving growing interest in their application to mathematical optimization. Beyond assisting with formulation, LLMs are increasingly expected to generate complete, executable optimization programs from business descriptions. However, optimization programs differ fundamentally from conventional scripts: they must satisfy hard domain constraints, interface correctly with mathematical solvers, and produce high-quality decisions under complex objectives. In enterprise production environments, these programs serve as reusable decision tools that must remain reliable as real-world data evolve. This raises a fundamental question: *can LLMs reliably generate optimization programs that satisfy hard constraints and achieve near-optimal decisions on unseen problem instances?*

Existing benchmarks have made notable progress in evaluating LLMs for optimization, ranging from natural-language formulation to solver-assisted reasoning (Ramamonjison et al., 2023; Huang et al., 2025b; AhmadiTeshnizi et al., 2024; Huang et al., 2025a; Astorga et al., 2025; Li et al., 2026; Zhou et al., 2026; Fu et al., 2026). However, conventional evaluations remain largely confined to *instance-level* assessments (Figure 1, left), testing whether a model finds a feasible solution for a single instance with fixed parameters. Such protocols fall short in two critical aspects: they cannot verify whether a synthesized program generalizes across changing data distributions without overfitting, and they typically rely on superficial checks—such as code executability or self-reported solver status—without independently verifying domain constraints. Consequently, they offer limited insight into whether LLMs can construct dependable optimization programs that strictly satisfy business rules and deliver near-optimal decisions on unseen data.

To address these limitations, we introduce TreasuryOpt, a benchmark designed to evaluate large language models on generating reliable optimization programs. We ground TreasuryOpt in enterprise

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.

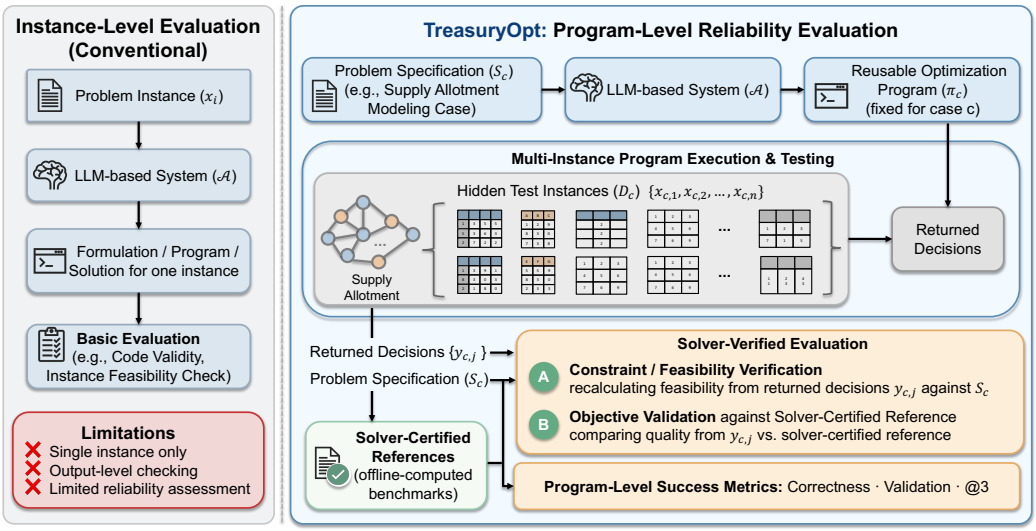


Figure 1: From conventional instance-level evaluation to program-level reliability benchmarking.

treasury operations—a mission-critical domain characterized by lexicographic multi-tier objectives, strict fund conservation, and dynamic settlement graphs (DAGs). In this setting, superficial heuristics or minor numerical discrepancies directly breach operational boundaries or trigger massive fund deficits, providing an exceptionally rigorous testbed for verifying whether synthesized programs preserve true business logic across varying environments.

TreasuryOpt evaluates the reliability of LLM-generated optimization programs through a standardized, solver-verified protocol (Figure 1, right). For each modeling case, an LLM-based system must synthesize a fixed, reusable optimization program directly from a textual problem specification. During execution, this fixed program is deployed across a suite of hidden test instances under strict computational budgets without further model intervention, spanning four enterprise treasury scenarios, 28 modeling cases, and 596 hidden test instances in total. Crucially, our protocol does not rely on model-reported solver logs; instead, it intercepts the returned decisions to independently recompute constraint feasibility from scratch and verifies objective optimality against solver-certified reference bounds, producing program-level Correctness and Validation metrics.

Extensive evaluations across six frontier LLMs—evaluated under standalone and agentic execution across solver-hidden and solver-aware conditions—reveal a persistent *feasibility–optimality gap*: systems face considerable difficulty satisfying domain constraints under single sampling (@1), with success rates collapsing further when requiring solver-certified optimality. Furthermore, while interactive coding harnesses improve solution quality via iterative self-debugging, they incur orders of magnitude higher generation costs with only limited performance gains and remain vulnerable to local self-testing blind spots. To understand the root causes of these failures, our systematic failure analysis demonstrates that basic syntax crashes are virtually negligible (under 4% across all settings); instead, programmatic failures stem from deeper mathematical and algorithmic bottlenecks, divided among domain constraint violations, execution timeouts, and suboptimal solution quality upon independent verification. These results demonstrate that reliable optimization program generation demands mathematical generalization and robust rule modeling far beyond surface-level code executability.

In summary, our contributions are threefold: (1) we introduce TreasuryOpt, a benchmark dedicated to benchmarking LLMs on generating reliable optimization programs for enterprise treasury operations; (2) we propose a program-level and solver-verified evaluation framework that assesses generated optimization programs through hidden instances and independent correctness checking; (3) we provide a comprehensive empirical evaluation of frontier LLMs across standalone and agentic settings, revealing a persistent feasibility–optimality gap and characterizing their vulnerabilities in generating reliable optimization programs.

2 Related Work

LLM-assisted Optimization Modeling. Large language models (LLMs) have recently been explored as assistants for mathematical optimization modeling, including formulation generation, optimization reasoning, and modeling-oriented data construction. Early benchmarks such as NL4OPT evaluate whether LLMs can translate natural-language optimization descriptions into formal formulations by identifying key components such as decision variables, objectives, and constraints (Ramamonjison et al., 2023). Subsequent studies, including MAMO, ORQA, and OptMATH, further investigate LLM capabilities in mathematical optimization reasoning and scalable optimization modeling through broader task collections and data synthesis frameworks (Huang et al., 2025b; Mostajabdaveh et al., 2025; Lu et al., 2025). These efforts demonstrate the potential of LLMs to bridge the gap between domain-level problem descriptions and formal optimization representations, enabling automated assistance in different stages of the optimization modeling process.

LLM-based Optimization Agents and Evaluation. Beyond optimization modeling, recent advances in large language model (LLM) agents have enabled models to interact with external tools, execution environments, and specialized software systems to solve complex tasks. Tool-integrated agents can decompose problems, invoke external resources, receive execution feedback, and iteratively refine their outputs through interaction with external environments (Yao et al., 2023; Schick et al., 2023; Qin et al., 2023; Guo et al., 2024). These capabilities have motivated the development of LLM-based optimization agents, where language models are combined with optimization solvers and programming environments to support automated model construction, solution generation, and decision-making workflows (AhmadiTeshnizi et al., 2024; Huang et al., 2025a; Li et al., 2026; Fu et al., 2026). Such systems extend LLMs beyond standalone formulation generation by enabling interactive optimization workflows involving solver feedback, code execution, and iterative refinement.

However, evaluating these systems remains challenging. Existing benchmarks for LLM agents and executable systems, such as HumanEval, SWE-bench, AgentBench, and ToolBench, emphasize execution-based correctness, task completion, or interaction capabilities (Chen et al., 2021; Jimenez et al., 2024; Liu et al., 2024; Qin et al., 2023). In optimization domains, existing evaluations mainly focus on individual capabilities, such as formulation accuracy, reasoning performance, solver interaction, or solution quality on specific instances (Ramamonjison et al., 2023; Huang et al., 2025b;a). These evaluations provide valuable insights into LLM optimization capabilities, but do not fully assess whether an LLM-based optimization system can construct reusable optimization programs that preserve domain requirements and remain reliable across diverse problem instances.

3 The TreasuryOpt Benchmark: Task Formulation and Design

3.1 Problem Formulation: Program-Level Reliability

Practical enterprise optimization problems are typically governed by fixed operational rules, while their numerical parameters fluctuate across operating environments. In such workflows, producing a one-off numerical solution for a single instance is insufficient; human practitioners instead require an optimization program that encapsulates the underlying decision logic and executes reliably across changing inputs. To benchmark this programmatic reliability, TreasuryOpt evaluates whether LLMs can synthesize reusable optimization code from abstract operational specifications.

Formally, the benchmark comprises a set of modeling cases $\mathcal{C}$. Each case $c \in \mathcal{C}$ is defined by a textual specification S_c detailing the operational rules, mathematical constraints, and input–output formats. Given only S_c, an LLM system $\mathcal{A}$ synthesizes an optimization program:

$$\pi_c = \mathcal{A}(S_c). \tag{1}$$

Once generated, the program is frozen and deployed across a set of hidden instances $D_c = \{x_{c,1}, \ldots, x_{c,n}\}$. For each incoming operational instance $x_{c,j}$ executes to output a decision $y_{c,j}$:

$$y_{c,j} = \pi_c(x_{c,j}). \tag{2}$$

Program generation is decoupled from execution: the model observes no hidden instances during generation, and no runtime LLM prompting or debugging is permitted during evaluation across D_c.

Table 1: Composition of TreasuryOpt.

Scenario	Cases	Inst.	Core Modeling Features
Cash Pool Concentration	8	163	Account roles; capacity limits; multi-objective returns
Settlement Limit	6	100	Optional settlement; reserve cap; lexicographic objectives
Settlement Sequencing	6	102	Dependencies; divisibility; chain limits
Supply Allotment Dividend	8	231	Flow balance; borrowing; debt links; acyclicity
Total	**28**	**596**	–

3.2 Benchmark Construction and Treasury Scenarios

Treasury Scenarios and Dataset Statistics. Table 1 summarizes the composition of TreasuryOpt. The benchmark contains 4 treasury optimization scenarios, 28 modeling cases, and 596 hidden test instances. In the table, **Inst.** denotes the number of hidden instances associated with each scenario. TreasuryOpt follows a three-level organization. A *scenario* represents a broad class of treasury decision problems. Within each scenario, a *modeling case* specifies a particular decision logic and serves as the unit for generating an optimization program. Each modeling case is further associated with multiple *hidden instances* that preserve the same underlying decision semantics while varying the instance-specific inputs. This hierarchy allows the same generated program to be evaluated under different conditions without changing the decision problem it is intended to implement.

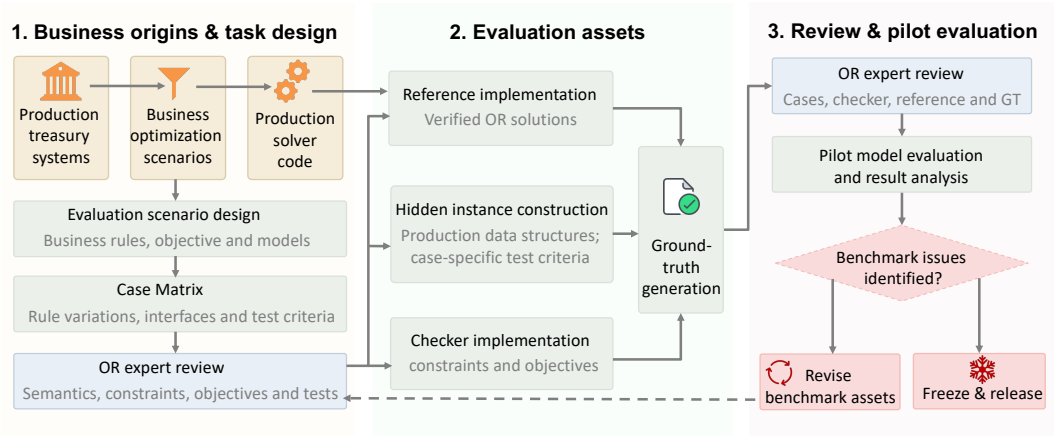


Figure 2: Overview of the TreasuryOpt benchmark construction pipeline.

Benchmark Construction Pipeline. As shown in Figure 2, the construction of TreasuryOpt follows a disciplined three-stage engineering pipeline:

- **Stage 1: Business Origins & Task Design.** We extract business optimization scenarios and production solver code from live enterprise treasury systems. These are formalized into standardized evaluation scenarios and an operational case matrix defining rule variations, interfaces, and test criteria, followed by initial OR expert review.
- **Stage 2: Evaluation Asset Generation.** For each modeling case, the pipeline curates three decoupled assets: (1) *Hidden instance construction*, synthesizing parametric operational instances that mirror production data structures and case-specific test criteria to stress-test domain rules under varying inputs; (2) *Reference implementation and solver-verified references*, precomputing fixed offline reference targets using verified formulations paired with the Gurobi optimizer to record certified feasibility statuses, strict numerical tolerances, and prioritized objective vectors for lexicographic comparison, serving as quality targets rather than rigid solution templates; and (3) *Checker implementation*, developing standalone Python checkers that operationalize a staged verification sequence covering execution, schema interface, feasibility status, hard business constraints, and recalculated objective bounds. Together, the reference models and hidden instances establish the ground truths, while the decoupled checkers form the independent verification harness.

- **Stage 3: Review & Pilot Evaluation.** Operations research experts conduct an end-to-end audit of all cases, checkers, reference models, and ground truths, followed by pilot model evaluation and discrepancy analysis. Any identified benchmark issues trigger an iterative asset revision loop before the benchmark suite is finalized, frozen, and released.

3.3 Solver-Verified Decoupled Evaluation Protocol

While verification executes over individual hidden instances, TreasuryOpt evaluates candidate programs at the modeling-case level. Crucially, our evaluation is completely decoupled from model-reported console logs or self-claimed solver termination statuses. Instead, the harness intercepts the returned decision variables $y_{c,j} = \pi_c(x_{c,j})$ and independently recalculates constraint satisfaction and objective quality from scratch against solver-certified reference bounds. For a modeling case c paired with hidden instance set $\mathcal{D}_c$, a candidate program π_c must satisfy the required checks across every instance in $\mathcal{D}_c$ to achieve case-level success. We report two primary criteria:

Correctness. A candidate program is *correct* for case c if it executes cleanly, conforms to schema interfaces, and strictly satisfies all domain hard constraints across all hidden instances:

$$\mathrm{Correct}(\pi_c) = \mathbb{I}\left[\forall x_{c,j} \in \mathcal{D}_c,\ \mathcal{V}_{\mathrm{corr}}(\pi_c, x_{c,j}) = 1\right], \tag{3}$$

where $\mathcal{V}_{\mathrm{corr}}$ denotes the collection of independent verification checks preceding objective evaluation.

Validation. A candidate program *validates* a case if it is correct and additionally matches the solver-certified reference objective within the prescribed tolerance on every hidden instance:

$$\mathrm{Valid}(\pi_c) = \mathbb{I}\left[\forall x_{c,j} \in \mathcal{D}_c,\ \mathcal{V}_{\mathrm{full}}(\pi_c, x_{c,j}) = 1\right], \tag{4}$$

where $\mathcal{V}_{\mathrm{full}}$ encompasses all verification stages. Validation enforces a strictly harder condition:

$$\mathrm{Valid}(\pi_c) = 1 \implies \mathrm{Correct}(\pi_c) = 1. \tag{5}$$

Sample Aggregation (@1 and @3). To account for generation variability across independent runs, we generate three candidate programs for each modeling case and report both single-attempt (@1) and best-of-three (@3) metrics. A case succeeds under an @3 metric if at least one candidate program satisfies the corresponding criterion:

$$\mathrm{Validation@3}(c) = \mathbb{I}\left[\max_{k\in\{1,2,3\}} \mathrm{Valid}(\pi_c^{(k)}) = 1\right], \tag{6}$$

with $\mathrm{Correctness@3}$ defined analogously. Dataset-level scores are obtained by averaging these indicators across all 28 modeling cases.

4 Experiments

4.1 Experimental Setup

Evaluated systems. We evaluate LLM-based systems under two settings: direct generation and harness-based execution. The evaluated systems use six LLMs: Fable 5, GPT-5.6 Sol, Qwen 3.8 Max, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash. All main models are evaluated with OpenClaw, and selected models are further compared with their native harnesses, including Codex, and Kimi Code. Each system receives the same problem specification and generates an optimization program following the benchmark interface.

Generation Settings. We consider two solver information settings: *Solver Hidden*, where no predefined solver menu is provided, and *Solver Menu*, where the available optimization tools are specified in the prompt. The generated program is fixed after generation and evaluated on all hidden instances associated with the case without additional model invocation.

Execution Environment. All generated programs are evaluated in a standardized environment with a common program interface and execution time limits. During hidden-test execution, candidates only receive the instance input and cannot access reference solutions or verifier outputs. Generation failures, runtime errors, and execution timeouts are retained as evaluation outcomes without manual correction. Further implementation details are provided in Appendix A.1.

Table 2: Overall evaluation results on TREASURYOPT across standalone and agentic systems.

Model	System	Solver Hidden				Solver Menu			
		V@1	C@1	V@3	C@3	V@1	C@1	V@3	C@3
Fable 5	Single LLM	3.57	28.57	7.14	28.57	17.86	46.43	50.00	67.86
	OpenClaw	21.43	42.86	39.29	75.00	32.14	**60.71**	**60.71**	71.43
GPT-5.6 Sol	Single LLM	7.14	14.29	7.14	25.00	10.71	42.86	32.14	64.29
	OpenClaw	3.57	28.57	14.29	46.43	10.71	32.14	35.71	53.57
Qwen 3.8 Max	Single LLM	7.14	50.00	14.29	67.86	3.57	28.57	28.57	64.29
	OpenClaw	**46.43**	**67.86**	53.57	**92.86**	14.29	39.29	57.14	78.57
GLM 5.3	Single LLM	10.71	25.00	14.29	39.29	14.29	39.29	17.86	42.86
	OpenClaw	14.29	46.43	39.29	85.71	17.86	42.86	50.00	**85.71**
Kimi K3	Single LLM	0.00	39.29	3.57	46.43	28.57	42.86	35.71	64.29
	OpenClaw	35.71	53.57	**64.29**	89.29	**35.71**	**60.71**	50.00	78.57
DeepSeek V4.1 Flash	Single LLM	3.57	21.43	3.57	32.14	10.71	25.00	25.00	50.00
	OpenClaw	32.14	57.14	46.43	78.57	21.43	42.86	50.00	67.86

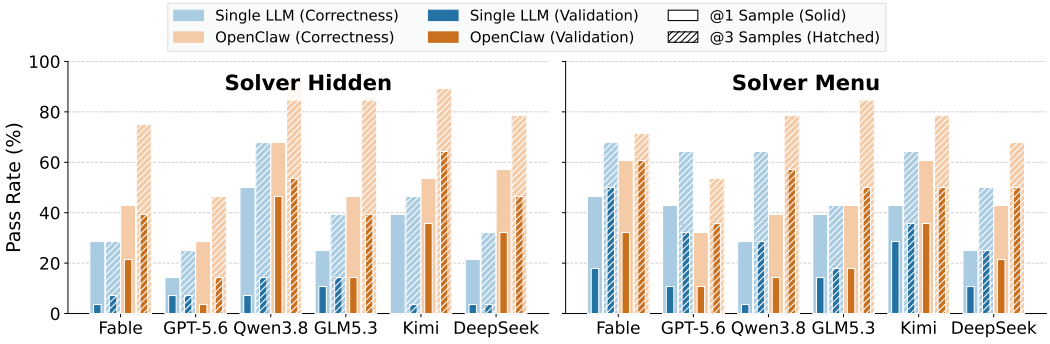


Figure 3: Overall performance visualization across standalone and agentic systems.

4.2 OVERALL PERFORMANCE

Table 2 and Figure 3 summarize the overall evaluation results across all 28 modeling cases in Treasury-Opt, contrasting direct generation against agentic execution across both solver information conditions. To ensure rigorous cross-model fairness under controlled environments, agentic evaluations in this main benchmark strictly employ a unified execution framework (OpenClaw) evaluated under identical tooling, context limits, and three-sample budgets. Proprietary native coding harnesses (e.g., Claude Code, Codex, and Kimi Code) are excluded from the main table due to asymmetric sampling budgets and incomplete model-family coverage, and are instead thoroughly investigated in Section 4.3. From these unified evaluations, we highlight three empirical findings:

Persistent Feasibility–Optimality Gap. Across all evaluated models and configurations, we observe a systematic and substantial discrepancy between operational Correctness C and solver-certified Validation V. As visually evidenced in Figure 3, this gap manifests as a stark contrast between the elevated constraint correctness (light bars) and the severely collapsed validation pass rates (dark bars). Under single-sample direct generation without solver hints, systems already face considerable difficulty satisfying domain constraints: C@1 hovers between 14.29% and 39.29% for five of the six frontier models (with only Qwen 3.8 Max reaching 50.00%). When requiring solver-certified optimality, success rates collapse further: three standalone models achieve V@1 $\leq$ 3.57%, with Kimi K3 validating zero cases. Even when agentic execution substantially improves constraint satisfaction under repeated sampling (C@3 reaching 85.71%–92.86% for GLM 5.3 and Qwen 3.8 Max), their corresponding V@3 scores drop to 39.29% and 53.57%. These empirical

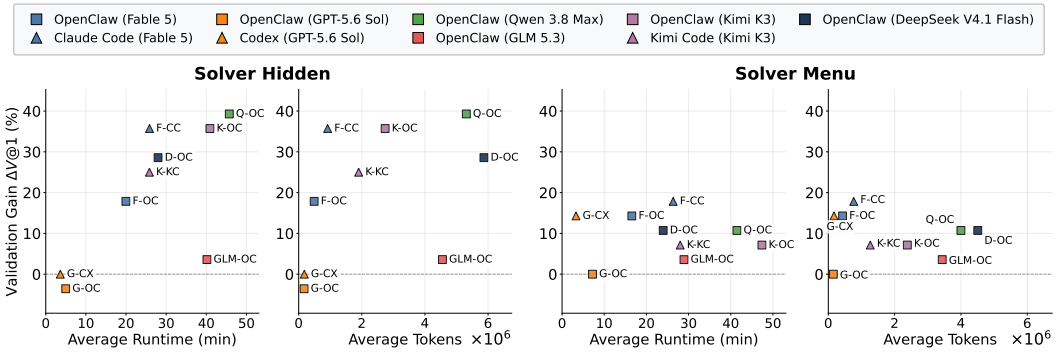


Figure 4: Agentic optimization performance on TreasuryOpt.

results show that generating constraint-satisfying programs does not mean the system understands how to balance trade-offs and reach mathematically optimal decisions.

Benefits of Repeated Sampling and Candidate Diversity. As shown by the shift from solid (@1) to hatched (@3) bars in Figure 3, repeated independent sampling significantly expands the solution coverage across all architectures. However, multi-sample aggregation also reshuffles relative model standings. For instance, under *Solver Hidden*, Qwen 3.8 Max leads on single-sample validation with 46.43% $V@1$, yet Kimi K3 overtakes it under repeated sampling by achieving 64.29% $V@3$. A similar ranking inversion occurs under *Solver Menu*, where Kimi K3 leads on $@1$ while Fable 5 attains the top $@3$ coverage. This gap between @1 and @3 shows that while repeated sampling increases the chance of finding a valid solution, individual generations remain brittle and inconsistent.

Divergent Impact of Solver Hints on Standalone vs. Agentic Systems. Explicitly providing a solver menu produces markedly different effects across execution paradigms, as visible in Figure 3. For standalone direct generation, the verified solver menu offers a consistent advantage across nearly all models: Kimi K3 improves from 0.00% to 28.57% in $V@1$, while Fable 5 gains 14.29 percentage points. In this regime, knowing available solvers eliminates uncertainty regarding runtime dependencies and helps models formulate valid solver APIs. Conversely, for agentic workflows under OpenClaw, the impact of the solver menu is non-monotonic. While Fable 5 validation improves by 10.71 percentage points, Qwen 3.8 Max drops sharply by 32.14 percentage points, falling from 46.43% to 14.29%. Because interactive agents can dynamically probe the runtime sandbox and self-test backends, pre-specifying available solvers does not offer a uniform benefit, and can induce syntactic restructuring and behavioral churn that disrupt otherwise viable formulation habits.

4.3 Agentic Optimization

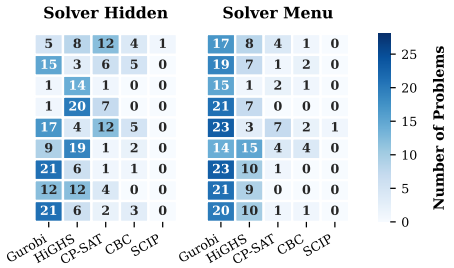
While agentic harnesses enhance programmatic validation, their practical viability depends on the trade-off between solution quality and generation overhead. Figure 4 examines this trade-off by plotting the validation gain ($\Delta V@1$) against resource consumption across both *Solver Hidden* (left panels) and *Solver Menu* (right panels) conditions. This evaluation encompasses nine agentic configurations across two paradigms: (1) the unified OpenClaw harness evaluated across six base models, denoted as F-OC, G-OC, Q-OC, GLM-OC, K-OC, and D-OC; and (2) three specialized native coding environments, namely Claude Code with Fable 5 (F-CC), Codex with GPT-5.6 Sol (G-CX), and Kimi Code with Kimi K3 (K-KC). Resource consumption is measured on a per-case synthesis basis along two complementary axes. *Average Runtime* records the elapsed wall-clock duration spanning LLM queries, tool invocations, code modifications, internal test runs, and idle network latencies, excluding downstream hidden-test verification. *Average Tokens* tracks the cumulative volume of input, cached, reasoning, and output tokens consumed across the full multi-turn trajectory for each case in agentic harnesses, compared against the single effective completion in direct synthesis.

Validation Gains at Elevated Computational Costs. As visualized in Figure 4, integrating models into coding harnesses yields validation gains ($\Delta V@1 > 0$) for the majority of setups, but simultaneously introduces noticeable generation overhead in both execution time and token consumption. The scatter plots illustrate a stark operational trade-off across harness paradigms: specialized native

Table 3: Behavioral churn and performance shifts across prompt conditions.

Model	System	Gurobi Cases		Val. Passes		Dynamics		ΔV (%)
		H	M	H	M	Gain	Drop	
Fable 5	OpenClaw	5	17 ↑	6	9 ↑	8	5	+10.71
	Claude Code	15	19 ↑	11	10 ↓	3	4	-3.57
GPT-5.6 Sol	OpenClaw	1	15 ↑	1	3 ↑	2	0	+7.14
	Codex	1	21 ↑	2	7 ↑	6	1	+17.86
Qwen 3.8 Max	OpenClaw	17	23 ↑	13	4 ↓	1	10	-32.14
GLM 5.3	OpenClaw	9	14 ↑	4	5 ↑	4	3	+3.57
Kimi K3	OpenClaw	21	23 ↑	10	10	3	3	0.00
	Kimi Code	12	21 ↑	7	10 ↑	5	2	+10.71
Deepseek v4.1 Flash	OpenClaw	21	20 ↓	9	6 ↓	2	5	-10.71

Figure 5: Number of modeling cases using each solver under Hidden and Menu conditions.



harnesses converge within tighter computational envelopes, whereas open-ended environments like OpenClaw trigger extensive multi-turn probing that can drive runtimes to 25–50 minutes and token expenditures into the millions.

Solver Hints and Validation Margins. Examining the vertical distribution in Figure 4 reveals a systematic disparity across prompt conditions: validation gains are visibly concentrated in higher ranges under *Solver Hidden* than under *Solver Menu*. This contrast reflects the disparate baseline capabilities established in Section 4.2. Under *Solver Hidden*, standalone direct generation struggles severely without runtime awareness, allowing agentic exploration to achieve substantial initial recoveries. Once available backends are explicitly disclosed in *Solver Menu*, the standalone baseline rises noticeably, which substantially narrows the marginal headroom available for interactive harnesses to deliver additional gains.

4.4 Solver Awareness Analysis

We next examine how providing explicit solver awareness affects the choice of optimization backends and downstream programmatic reliability. Figure 5 counts modeling cases whose generated programs invoke each solver during hidden-test execution (at most once per case and solver; a case may use multiple solvers), while Table 3 tracks the resulting behavioral churn and validation shifts between *Solver Hidden* (H) and *Solver Menu* (M). Notably, the vertical axis of Figure 5 directly mirrors the evaluated model–harness rows in Table 3, enabling direct row-by-row inspection between backend selection and performance transitions. Here, validation passes (val. passes) count the number of modeling cases that satisfy all domain constraints and achieve certified optimality. To quantify formulation stability beyond aggregate totals, Table 3 tracks case-level validation dynamics: *Gain* measures previously failed cases that successfully validate under the menu condition, while *Drop* captures previously validated cases that fail once the solver menu is provided.

Emergence of Solver Preference Under Awareness. As visualized in Figure 5, backend selection exhibits marked fragmentation under *Solver Hidden*, scattering across various open-source and constraint-programming engines (including HiGHS, CP-SAT, and CBC). In contrast, introducing the verified solver menu (*Solver Menu*) triggers a pronounced consolidation toward commercial-grade engines. Across the evaluated model–harness setups in Table 3, eight of nine setups use Gurobi on more modeling cases under Menu than under Hidden. Although select models retain secondary usage of open-source tools, disclosing viable backends clearly steers modern LLMs toward commercial-grade optimization engines whenever available.

Behavioral Churn and Performance Divergence. Crucially, increased adoption of commercial solvers does not monotonically translate into higher validation success. Across the nine setups in Table 3, five improve in validation rate, one remains unchanged, and three experience performance degradations. Tracking case-level transitions reveals substantial underlying behavioral churn: disclosing viable solvers prompts models to restructure their formulation templates and API syntax (e.g., migrating from open-source formulations to commercial solver APIs). When these syntactic alterations introduce subtle modeling errors, the resulting formulation bugs can negate the computational advantages of the underlying solver engine.

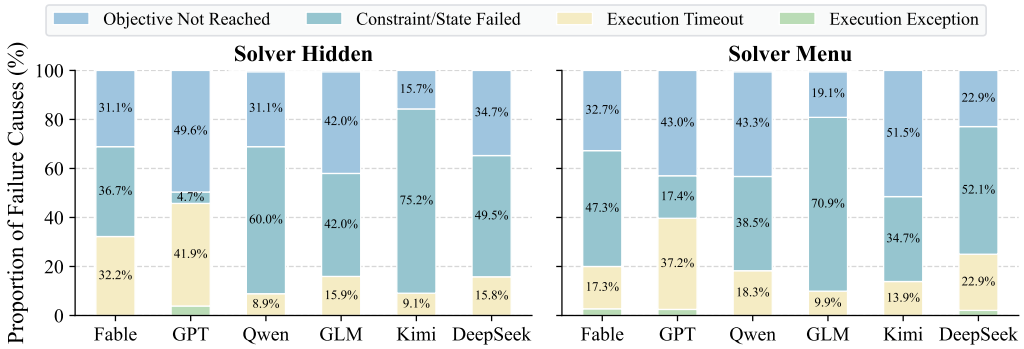


Figure 6: Failure cause distribution across LLMs under OpenClaw.

4.5 Failure Analysis

To uncover the bottlenecks of programmatic optimization reliability, Figure 6 presents a systematic breakdown of failure causes across all evaluated models under the unified agentic harness (OpenClaw) across both *Solver Hidden* and *Solver Menu* conditions. Across the 596 hidden test instances per model, failed attempts are partitioned into four mutually exclusive categories along our sequential verification pipeline: (1) *code execution exception*, where the script crashes due to runtime or interface errors; (2) *execution timeout*, where the program exceeds the outer execution time limit, potentially due to computationally inefficient formulations or solution algorithms; (3) *constraint/state failure*, where the returned decision violates physical operational constraints or misreports feasibility; and (4) *objective not reached*, where a fully feasible solution fails to match the solver-certified optimal reference bounds.

A prominent observation is that execution exception represents a negligible fraction of total failures, remaining barely visible across all evaluated models and prompt conditions. This indicates that modern frontier LLMs have mastered surface-level programming syntax, solver library imports, and execution interfaces. Instead, the distribution of programmatic failures exposes a noticeable behavioral divergence across architectures: On one hand, GPT-5.6 Sol is primarily bottlenecked by suboptimal solutions (43.0% to 49.6% of failures) alongside substantial execution timeouts (37.2% to 41.9%). On the other hand, open-weights and reasoning-oriented architectures, including Qwen 3.8 Max, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash, exhibit low timeout rates but are predominantly bottlenecked by constraint and state violations. While both behaviors ultimately prevent successful deployment, the former is characterized by long-running executions that exceed computational limits, whereas the latter is marked by rapid terminations that fail independent constraint recalculation.

5 Conclusion

In this work, we introduce TreasuryOpt, a benchmark evaluating large language models on generating reliable optimization programs that remain robust across unseen instances. Covering four enterprise treasury scenarios, 28 modeling cases, and 596 hidden test instances, TreasuryOpt shifts the evaluation paradigm from one-off instance solving to program-level reliability, decoupling verification from model outputs by independently recalculating domain constraints and objective optimality against certified solver bounds. Extensive evaluations across frontier LLMs expose a persistent feasibility–optimality gap: systems face severe difficulty satisfying physical domain constraints, with success collapsing further when enforcing solver-certified optimality. While interactive agentic harnesses improve validation through iterative self-debugging, they incur steep computational costs without eliminating core mathematical modeling failures. These findings demonstrate that reliable optimization program generation demands capabilities far beyond code synthesis and tool execution. TreasuryOpt provides a solver-verified foundation for evaluating and advancing LLM-based optimization systems, while future work will expand its coverage to broader enterprise domains, dynamic multi-period settings, and richer operational constraints.

AI USE STATEMENT

LLMs were used for auxiliary support, including literature search assistance, grammatical editing, writing refinement, and code debugging.

ETHICS STATEMENT

TreasuryOpt is constructed from abstracted enterprise optimization scenarios without exposing confidential business information or private data. The benchmark is intended for research evaluation, and LLM-generated optimization programs should be subject to domain expert review and independent validation before practical deployment.

REPRODUCIBILITY STATEMENT

We provide detailed descriptions of TreasuryOpt, including benchmark construction, evaluation protocol, experimental settings, and implementation details. The benchmark instances, verification framework, and evaluation code will be released to enable independent reproduction and further research.

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*, ICML'24. JMLR.org, 2024.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Yongchang Fu, Xinjie Huang, Chengjun Dai, Chengzhe Feng, Junshao Zhang, and Hong Zhu. Optiagent-bench: Benchmarking end-to-end optimization r&d agents on real-world business problems. *arXiv preprint arXiv:2607.10768*, 2026.

Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, 2024.

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*, 73(6):2986–3009, 2025a. ISSN 1526-5463. doi: 10.1287/opre.2024.1233. URL http://dx.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.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Jiajun Li, Mingshu Cai, Yixuan Li, Yu Ding, Ran Hou, Guanyu Nie, Xiongwei Han, and Wanyuan Wang. Oragentbench: Can llm agents solve challenging operations research tasks end to end? *arXiv preprint arXiv:2606.19787*, 2026.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

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, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/lu25o.html.

Mahdi Mostajabdaveh, Timothy Tin Long Yu, Samarendra Chandan Bindu Dash, Rindra Ramamonjison, Jabo Serge Byusa, Giuseppe Carenini, Zirui Zhou, and Yong Zhang. Evaluating llm reasoning in the operations research domain with orqa. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 24902–24910, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

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.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Chenyu Zhou, Xinyun Lu, Jiangyue Zhao, Jianghao Lin, Dongdong Ge, and Yinyu Ye. Orspace: A full-lifecycle workspace benchmark for industrial optimization agents. *arXiv preprint arXiv:2605.28158*, 2026.

A Appendix

A.1 Experimental Details

This appendix summarizes the experimental configurations used in the evaluation, including benchmark coverage, generation settings, runtime environment, and reporting protocol.

A.1.1 Benchmark and Experimental Coverage

The benchmark consists of four treasury optimization scenarios with 28 cases and 596 hidden test instances. The main evaluation includes Fable 5, GPT-5.6 Sol, Qwen 3.8 Max, GLM 5.3, Kimi K3, and DeepSeek V4.1 Flash. The experiments study three factors: (1) model variation under a unified harness (OpenClaw), (2) solver information provided to the model (Solver Hidden vs. Solver Menu), and (3) execution setting (single generation, OpenClaw, and native harnesses).

A.1.2 Generation Configuration

The experiments vary two aspects of generation configuration: (1) execution mode and (2) solver-information condition. The corresponding settings are summarized in Table 4.

The main harness versions are OpenClaw 2026.6.11, Codex CLI 0.147.0, Claude Code 2.1.114, and Kimi Code 0.36.1. During generation, models do not access hidden test results. All groups use the

Table 4: Generation settings.

Category	Setting	Description
Execution mode	Single generation	A single model call generates the program without code tools or interactive execution.
Execution mode	Harness	The model uses tools, executes code, and modifies programs before submitting the final program.
Solver information	Solver Hidden	The prompt does not provide a solver menu; solver usage is not disabled.
Solver information	Solver Menu	The prompt provides the verified available solver menu without specifying priority or requiring a specific solver.

original task specification without additional modeling hints. No sub-agent collaboration is enabled, and native harness system prompts are retained.

The generation budgets and sampling settings are summarized in Table 5. Single-generation runs encountering retryable external errors may be retried or resumed. No unified maximum retry count is imposed; retry behavior follows the recorded configuration of each experimental batch.

Table 5: Generation budgets and sampling settings.

Execution mode	Generation budget	Samples per case	Interaction setting
Single generation	1,800s per request	3	No tool interaction
OpenClaw	10,800s per agent run	3	No outer agent round limit
Claude Code / Codex / Kimi Code	10,800s per agent run	3	No outer agent round limit

All OpenClaw experiments use a context window of 200,000 tokens. The model reasoning configurations and output limits used in the @1 experiments are shown in Table 6.

A.1.3 Evaluation Configuration

OpenClaw Hidden and Menu experiments use fixed execution limits for each hidden instance, as summarized in Table 7. The two limits serve different purposes. The solver time limit is controlled by the generated program according to the problem specification, while the outer execution time limit is controlled by the evaluator and covers the complete processing of one hidden instance, including modeling, solving, and result processing.

In this paper, execution timeout refers specifically to exceeding the outer execution limit and is distinct from solver-level timeouts. Program generation time is excluded from both limits. The OpenClaw Hidden and Menu experiments are executed under Linux x86_64 with CPython 3.11.13. Gurobi is used with a valid license.

A.1.4 Evaluation and Reporting Protocol

Quality is measured at the modeling-case level. The denominator is 28 cases for overall results and the number of cases in each scenario for scenario-level results. A case is counted as successful only when all hidden test instances associated with that case pass.

Correctness (C) and Validation (V) follow the definitions in Section 3.3. When the ground truth (GT) is infeasible, a program passes both C and V if it executes successfully and returns `status="infeasible"`. Objective values are not compared in this case. Timeout or other returned statuses, such as `time_limit`, do not pass.

@1 reports the first sampling result. @3 reports whether at least one of the three main samples passes the entire case. Multiple modification rounds within one agent run are counted as a single sample. Confirmed generation failures are counted as failures, while unresolved results are not directly counted as failures. Unless explicitly marked as @3, quality comparisons, failure analysis, and generation cost analysis use first-sample results.

Table 6: Model reasoning configurations and output limits.

Model	Runtime	Reasoning configuration	Output limit
Fable 5	Single generation / OpenClaw	`output_config.effort=high`	64,000
	Claude Code	Adaptive thinking, `output_config.effort=high`	32,000 (CLI reported)
GPT-5.6 Sol	Single generation / Codex	`reasoning_effort=medium`	Not explicitly set
	OpenClaw	`reasoning_effort=medium`	64,000
Qwen 3.8 Max	Single generation	`enable_thinking=true`	Not explicitly set
	OpenClaw	`enable_thinking=true`	64,000
GLM 5.3	Single generation / OpenClaw	`reasoning_effort=high`	128,000
Kimi K3	Single generation / Kimi Code	`reasoning_effort=max`	1,048,576
	OpenClaw	`reasoning_effort=max`	64,000
DeepSeek V4.1 Flash	Single generation	`enable_thinking=true`, `reasoning_effort=high`	Not explicitly set
	OpenClaw	`thinking=high`; request overrides include `enable_thinking` and `reasoning_effort`	64,000
Opus 5	Claude Code	Adaptive thinking, `output_config.effort=xhigh`	32,000 (CLI reported)

Table 7: Hidden-test execution limits.

Scenario	Solver time limit	Outer execution time limit
Cash Pool	30 seconds	60 seconds
Limit	Case 01–04: 180 seconds; Case 05–06: 210 seconds	300 seconds
Sequencing	75 seconds	105 seconds
Supply Dividend	50 seconds	75 seconds

Generation cost is reported using three averages:

- **Generation time**: wall-clock time including model requests, tool calls, code modification, self-testing, and waiting time, excluding hidden-test evaluation.
- **Rounds**: the number of logical model requests issued by the main agent, excluding tool calls.
- **Token usage**: the sum of input and output tokens, counting cached input and reasoning tokens once.

For single-generation experiments, token usage is measured from the final effective response. For harness experiments, token usage is aggregated over all recorded rounds. Groups without complete token accounting do not report average token usage.

A.2 Experimental Results

This subsection provides comprehensive supplementary results and fine-grained empirical breakdowns across scenarios, generation budgets, reasoning configurations, and suboptimality distributions, complementing the findings presented in Section 4.

Table 8: Detailed evaluation results on TreasuryOpt, broken down by scenarios.

Model	System	Solver Hidden					Solver Menu				
		$V@1$	$C@1$	Δ_{C-V}	$V@3$	$C@3$	$V@1$	$C@1$	Δ_{C-V}	$V@3$	$C@3$
Scenario: Cash Pool Concentration											
Fable 5	Single LLM	12.50	25.00	12.50	25.00	25.00	0.00	12.50	12.50	25.00	25.00
	OpenClaw	12.50	25.00	12.50	25.00	62.50	0.00	25.00	25.00	0.00	25.00
GPT-5.6 Sol	Single LLM	25.00	25.00	0.00	25.00	25.00	25.00	50.00	25.00	50.00	75.00
	OpenClaw	12.50	12.50	0.00	25.00	25.00	12.50	12.50	0.00	37.50	50.00
	Codex	25.00	25.00	0.00	37.50	37.50	12.50	12.50	0.00	62.50	75.00
Qwen 3.8 Max	Single LLM	12.50	25.00	12.50	25.00	62.50	0.00	25.00	25.00	37.50	75.00
	OpenClaw	87.50	87.50	0.00	87.50	87.50	12.50	37.50	25.00	50.00	75.00
GLM 5.3	Single LLM	12.50	12.50	0.00	12.50	12.50	12.50	25.00	12.50	25.00	25.00
	OpenClaw	25.00	50.00	25.00	37.50	62.50	0.00	0.00	0.00	50.00	50.00
Kimi K3	Single LLM	0.00	12.50	12.50	0.00	12.50	12.50	12.50	0.00	25.00	25.00
	OpenClaw	12.50	25.00	12.50	50.00	75.00	12.50	37.50	25.00	25.00	50.00
	Kimi Code	12.50	62.50	50.00	12.50	62.50	12.50	12.50	0.00	25.00	25.00
DeepSeek V4.1 Flash	Single LLM	12.50	12.50	0.00	12.50	12.50	0.00	0.00	0.00	12.50	12.50
	OpenClaw	12.50	25.00	12.50	25.00	37.50	0.00	12.50	12.50	25.00	25.00
Scenario: Settlement Limit											
Fable 5	Single LLM	0.00	66.67	66.67	0.00	66.67	0.00	50.00	50.00	50.00	83.33
	OpenClaw	33.33	83.33	50.00	50.00	83.33	33.33	83.33	50.00	83.33	83.33
GPT-5.6 Sol	Single LLM	0.00	33.33	33.33	0.00	66.67	0.00	66.67	66.67	16.67	83.33
	OpenClaw	0.00	66.67	66.67	33.33	100.00	0.00	66.67	66.67	50.00	83.33
	Codex	0.00	66.67	66.67	0.00	66.67	33.33	50.00	16.67	66.67	66.67
Qwen 3.8 Max	Single LLM	0.00	83.33	83.33	0.00	83.33	16.67	33.33	16.66	33.33	66.67
	OpenClaw	33.33	50.00	16.67	33.33	83.33	16.67	66.67	50.00	50.00	66.67
GLM 5.3	Single LLM	0.00	33.33	33.33	0.00	83.33	16.67	33.33	16.66	16.67	50.00
	OpenClaw	16.67	83.33	66.66	50.00	100.00	50.00	100.00	50.00	83.33	100.00
Kimi K3	Single LLM	0.00	83.33	83.33	0.00	100.00	16.67	50.00	33.33	16.67	83.33
	OpenClaw	16.67	33.33	16.66	50.00	83.33	16.67	66.67	50.00	33.33	83.33
	Kimi Code	16.67	50.00	33.33	33.33	83.33	33.33	33.33	0.00	50.00	83.33
DeepSeek V4.1 Flash	Single LLM	0.00	33.33	33.33	0.00	66.67	16.67	50.00	33.33	33.33	66.67
	OpenClaw	33.33	66.67	33.34	66.67	83.33	33.33	50.00	16.67	50.00	83.33
Scenario: Settlement Sequencing											
Fable 5	Single LLM	0.00	33.33	33.33	0.00	33.33	16.67	66.67	50.00	50.00	83.33
	OpenClaw	50.00	66.67	16.67	66.67	83.33	33.33	50.00	16.67	66.67	83.33
GPT-5.6 Sol	Single LLM	0.00	0.00	0.00	0.00	16.67	0.00	33.33	33.33	16.67	66.67
	OpenClaw	0.00	50.00	50.00	0.00	66.67	0.00	16.67	16.67	33.33	50.00
	Codex	0.00	33.33	33.33	33.33	50.00	16.67	16.67	0.00	16.67	50.00
Qwen 3.8 Max	Single LLM	16.67	83.33	66.66	16.67	100.00	0.00	50.00	50.00	33.33	83.33
	OpenClaw	16.67	66.67	50.00	33.33	100.00	16.67	50.00	33.33	16.67	66.67
GLM 5.3	Single LLM	33.33	66.67	33.34	50.00	83.33	33.33	83.33	50.00	33.33	83.33
	OpenClaw	16.67	66.67	50.00	50.00	100.00	16.67	33.33	16.66	50.00	100.00
Kimi K3	Single LLM	0.00	66.67	66.67	16.67	83.33	16.67	50.00	33.33	33.33	66.67
	OpenClaw	50.00	66.67	16.67	66.67	100.00	50.00	66.67	16.67	50.00	100.00
	Kimi Code	50.00	83.33	33.33	50.00	83.33	33.33	100.00	66.67	50.00	100.00
DeepSeek V4.1 Flash	Single LLM	0.00	50.00	50.00	0.00	66.67	16.67	33.33	16.66	16.67	50.00
	OpenClaw	50.00	83.33	33.33	66.67	100.00	33.33	83.33	50.00	83.33	83.33
Scenario: Supply Allotment Dividend											
Fable 5	Single LLM	0.00	0.00	0.00	0.00	0.00	50.00	62.50	12.50	75.00	87.50
	OpenClaw	0.00	12.50	12.50	25.00	75.00	62.50	87.50	25.00	100.00	100.00
GPT-5.6 Sol	Single LLM	0.00	0.00	0.00	0.00	0.00	12.50	25.00	12.50	37.50	37.50
	OpenClaw	0.00	0.00	0.00	0.00	12.50	25.00	37.50	12.50	25.00	37.50
	Codex	0.00	0.00	0.00	0.00	0.00	37.50	37.50	0.00	75.00	75.00
Qwen 3.8 Max	Single LLM	0.00	25.00	25.00	12.50	37.50	0.00	12.50	12.50	12.50	37.50
	OpenClaw	37.50	62.50	25.00	50.00	100.00	12.50	12.50	0.00	100.00	100.00
GLM 5.3	Single LLM	0.00	0.00	0.00	0.00	0.00	0.00	25.00	25.00	0.00	25.00
	OpenClaw	0.00	0.00	0.00	25.00	87.50	12.50	50.00	37.50	25.00	100.00
Kimi K3	Single LLM	0.00	12.50	12.50	0.00	12.50	62.50	62.50	0.00	62.50	87.50
	OpenClaw	62.50	87.50	25.00	87.50	100.00	62.50	75.00	12.50	87.50	87.50
	Kimi Code	25.00	50.00	25.00	75.00	87.50	62.50	75.00	12.50	100.00	100.00
DeepSeek V4.1 Flash	Single LLM	0.00	0.00	0.00	0.00	0.00	12.50	25.00	12.50	37.50	75.00
	OpenClaw	37.50	62.50	25.00	37.50	100.00	25.00	37.50	12.50	50.00	87.50

Table 9: Detailed breakdown of failure stages across models and prompt conditions.

Model	System	Solver Hidden					Solver Menu				
		Exc.	Timeout	Constr.	Subopt.	**Total**	Exc.	Timeout	Constr.	Subopt.	**Total**
Fable 5	OpenClaw	0	29	33	28	90	3	19	52	36	110
	Claude Code	0	22	48	16	86	0	16	50	30	96
GPT-5.6 Sol	OpenClaw	5	54	6	64	129	3	45	21	52	121
	Codex	3	44	19	74	140	2	36	9	42	89
Qwen 3.8 Max	OpenClaw	0	8	54	28	90	0	19	40	45	104
GLM 5.3	OpenClaw	0	22	58	58	138	0	14	100	27	141
Kimi K3	OpenClaw	0	11	91	19	121	0	14	35	52	101
	Kimi Code	0	21	30	55	106	0	26	61	16	103
DeepSeek V4.1 Flash	OpenClaw	0	15	47	33	95	2	22	50	22	96

Table 10: Distribution of violated operational hard constraints across treasury scenarios.

Scenario	Violated Constraint Category	Solver Hidden		Solver Menu	
		Violations	**Programs**	**Violations**	**Programs**
Cash Pool	Inflow capacity cap exceeded	140	24	182	28
	Disbursing account balance deficit	100	20	150	24
	Repayment exceeding current position	82	23	83	25
	Residual uncollected balance in full sweep	25	12	34	17
	Outflow exceeding cumulative inflow	19	5	46	13
	Priority debt repayment neglected	16	5	37	6
	Invalid inter-account routing	30	2	6	2
Settlement Limit	DAG layer balance overdraft	44	3	19	3
	Batch settlement negative balance	9	1	5	3
	Invoice split count exceeded	0	0	1	1
Sequencing	DAG layer balance overdraft	82	6	92	6
	Settlement chain length exceeded	1	1	9	3
	Invoice split count exceeded	0	0	6	2
	Circular execution / transfer path	0	0	2	1
Supply Dividend	Cyclic flow network (cycle formed)	0	0	4	4
	Deficit account unmet target balance	2	2	2	2
	Idle account overdraft	1	1	2	2

Table 11: Quantitative objective discrepancy against solver-certified reference targets (z^*) and feasibility classification errors.

Scenario	Objective	GT (z^*)	Unit	Solver Hidden			Solver Menu		
				Count	**Mean**	**Median**	**Count**	**Mean**	**Median**
Cash Pool	Uncollected Balance	Non-zero	%	56	809.19%	12.09%	66	617.76%	0.07%
	Uncollected Balance	Zero	Amount	37	11,463.87	0.06	49	6,375.90	0.06
	Net Interest Yield	Non-zero	%	32	11.45%	4.99%	8	15.11%	14.98%
Settlement Limit	Settled Volume	Non-zero	%	123	34.95%	10.77%	93	52.79%	47.66%
	Reserve Buffer	Non-zero	%	5	12.41%	5.41%	9	6.33%	5.25%
Sequencing	Required Funding	Non-zero	%	91	49.74%	31.97%	75	36.85%	28.49%
Supply Dividend	Total Transfer Cost	Non-zero	%	31	90.99%	55.55%	22	132.67%	74.04%

Feasibility Status Misclassification Category	Hidden (Occurrences / Progs)	Menu (Occurrences / Progs)
False Infeasibility (feasible GT, returned `infeasible`)	27 times / 15 programs	11 times / 7 programs
Infeasible Status Mismatch (infeasible GT, returned other statuses)	4 times / 4 programs	1 time / 1 program

A.2.1 SCENARIO-LEVEL BREAKDOWN AND DOMAIN HETEROGENEITY

Table 8 presents the detailed scenario-level pass rates across all 28 modeling cases. To uncover the structural and algorithmic bottlenecks underlying these results, we jointly inspect the multi-stage failure allocations across candidate programs (Table 9), the distribution of breached physical domain rules (Table 10), and the quantitative discrepancies relative to solver-certified reference targets (Table 11). Together, these evaluations reveal pronounced heterogeneity across operational environments:

Cash Pool Concentration (8 cases, 163 instances). Cash Pool exhibits the lowest validation pass rates in direct generation, with four of six models achieving $V@1 \leq 12.50\%$ under *Solver Hidden*. As verified in Table 10, this vulnerability is heavily driven by simultaneous multi-account balance limits and capacity bounds: exceeding net incoming capacity (breached in up to 28 programs) and disbursing balance deficits (up to 24 programs) constitute the most widespread operational violations. Furthermore, Table 11 demonstrates that even when generated programs satisfy all feasibility bounds, uncollected cash balances deviate from the certified zero target by thousands of currency units (mean gap exceeding 6,000 to 11,000 units), indicating that models routinely sacrifice Tier-1 sweeping objectives.

Settlement Limit Management (6 cases, 100 instances). Table 8 displays a pronounced feasibility–optimality gap in Settlement Limit, where models reliably achieve 50%–83% correctness ($C@1$) but consistently collapse on solver-certified validation (0% $V@1$ across all standalone models under *Solver Hidden*). This divide is corroborated by Table 9, where suboptimality (*Subopt.*) and timeouts dominate failures over basic syntax crashes. As shown in Table 11, feasible candidate programs miss reference cleared volumes by a median relative gap of 10.77% to 47.66%, confirming that models construct feasible settlement subsets but struggle to discover optimal transaction selections within tight intraday reserve caps.

Settlement Sequencing (6 cases, 102 instances). Governed by temporal precedence dependencies over directed acyclic graphs (DAGs), Settlement Sequencing exhibits high capability variance across model architectures. Failure inspection in Table 10 reveals that constraint violations are heavily concentrated in DAG-layer fund overdrafts (affecting 6 programs across 82 to 92 test instances), where concurrent batch settlements exhaust available account balances before subsequent receipts are processed. Moreover, Table 11 shows that feasible sequences still incur median funding requirement gaps of roughly 30%, reflecting difficulties in optimizing chronological settlement orders.

Supply Allotment Dividend (8 cases, 231 instances). Supply Dividend demonstrates extreme sensitivity to backend solver awareness: standalone models fail entirely under *Solver Hidden* (0% $V@1$), yet experience dramatic validation surges upon receiving explicit solver menus (e.g., Fable 5 reaching 75% $V@3$). Table 9 confirms that solver guidance substantially curbs execution timeouts and constraint failures in this domain. While programs under *Solver Menu* occasionally introduce structural cyclic flows (Table 10), providing explicit solver APIs enables models to reliably instantiate network-flow balance constraints, partially bridging the suboptimality gaps reported in Table 11.

A.2.2 COMPARISON BETWEEN OPENCLAW AND NATIVE HARNESSES

We further compare OpenClaw with native coding harnesses for three representative models: Fable 5, GPT-5.6 Sol, and Kimi K3. Table 12 reports correctness, validation performance, and generation costs under both Solver Hidden and Solver Menu settings.

Compared with OpenClaw, native harnesses improve validation performance for Fable 5 and GPT-5.6 Sol under both solver information settings, while Kimi K3 shows mixed effects: lower validation under Solver Hidden and comparable performance under Solver Menu. The computational overhead varies across systems. Fable 5 incurs higher generation time, interaction rounds, and token consumption with Claude Code, whereas GPT-5.6 Sol reduces generation time and rounds with Codex but has similar token usage. Kimi Code reduces both runtime and interaction rounds compared with OpenClaw, with additional token savings under Solver Menu.

A.2.3 TRAJECTORY ANALYSIS OF AGENTIC SELF-DEBUGGING

To investigate the microscopic behaviors of interactive coding harnesses, we analyze the execution and debugging trajectories across 504 first-sample candidate programs synthesized within OpenClaw

Table 12: Comparison between OpenClaw and native coding harnesses across solver prompt conditions. Generation succeeded for all 28 cases in every configuration (28/28); this denotes a saved program, not successful execution.

Prompt	Model	System	Quality		Generation Costs		
			C@1	V@1	Time (min)	Rounds	Tokens ($\times 10^4$)
Hidden	Fable 5	OpenClaw	42.86%	21.43%	19.97	13.71	49.00
		Claude Code	53.57%	39.29%	25.86	17.29	91.69
	GPT-5.6 Sol	OpenClaw	28.57%	3.57%	4.97	12.93	17.15
		Codex	28.57%	7.14%	3.63	5.21	17.75
	Kimi K3	OpenClaw	53.57%	35.71%	40.91	43.71	273.94
		Kimi Code	60.71%	25.00%	25.83	24.75	190.00†
Menu	Fable 5	OpenClaw	60.71%	32.14%	16.55	10.79	43.07
		Claude Code	57.14%	35.71%	26.35	15.89	76.66
	GPT-5.6 Sol	OpenClaw	32.14%	10.71%	7.22	12.57	15.09
		Codex	28.57%	25.00%	3.31	4.36	17.17
	Kimi K3	OpenClaw	60.71%	35.71%	47.39	36.25	238.23
		Kimi Code	53.57%	35.71%	28.01	25.71	126.50

and native coding environments. Behavioral events (e.g., test suite generation, runtime diagnostics, and constraint assertions) are parsed via automated pattern extraction and cross-verified against raw model invocation logs. Our qualitative inspection reveals a dual nature of agentic self-debugging:

Pathologies: Rule Reinforcement and Incomplete Input Coverage. While interactive environments allow models to iterate on generated code, internal unit tests frequently inherit the model's own misunderstandings of domain rules or fail to capture the distribution of operational inputs. We identify two representative pathological behaviors:

- *Adapting Test Assertions to Flawed Logic*: In Cash Pool Case 08 under *Solver Menu*, Kimi K3 (deployed within Kimi Code) synthesized an initial program that assigned an inflow account to receive 50 currency units while disbursing 80 units, fundamentally violating fund conservation. Instead of repairing the underlying formulation deficits, the agent iteratively modified its local test assertions, revising the expected return values to match the erroneous program outputs. Consequently, the finalized program clean-passed all local unit tests but immediately failed domain balance checks during decoupled evaluation.
- *Generalization Failures Beyond Local Toy Inputs*: In Cash Pool Case 07 under *Solver Menu*, Fable 5 (within Claude Code) successfully rectified identified pool capacity and risk threshold bugs against its local test script. Although the agent declared the implementation complete after all local assertions passed, downstream execution across the 21 hidden test instances encountered severe execution timeouts and suboptimal objective returns. This demonstrates that passing local synthetic test cases does not guarantee mathematical scalability or distributional robustness across unseen parametric instances.

Efficacy: Dynamic Resolution of Precision and Objective Discrepancies. Conversely, when agents establish faithful diagnostic routines, self-debugging effectively resolves subtle numerical and structural formulation defects that typically derail single-shot generation. Across trajectories, models construct diverse verification mechanisms—including floating-point boundary stress-testing, programmatic balance recalculation, randomized instance fuzzing, and dual-model objective cross-validation:

- *Sensitivity Analysis and Numerical Precision Tuning*: In Settlement Sequencing Case 04 under *Solver Hidden*, Fable 5 (within Claude Code) evaluated whether relaxing bill-splitting bounds would reduce reserve funding. Upon observing an anomalous increase of 0.01 in reserve requirements, the agent diagnosed numerical instability in solver tolerances, adjusted the precision configuration,

and re-executed validation tests. The resulting program achieved both Correctness (C) and certified Validation (V) across all hidden instances.

- *Boundary Condition Stress-Testing and Rounding Correction*: In Cash Pool Case 01 under *Solver Hidden*, Kimi K3 (within Kimi Code) constructed an adversarial test case with an account balance of 33.335. The test captured an illegal disbursement of 33.34 caused by floating-point rounding errors. By refining the internal rounding logic and re-testing against the boundary case, the agent eliminated the defect, allowing the final program to successfully satisfy all domain constraints and optimality bounds.

These empirical observations explain why agentic harnesses expand feasible coverage while remaining vulnerable to localized blind spots: iterative self-debugging effectively resolves deterministic numerical bugs, but cannot replace external solver-certified verification when the agent harbors deep business rule misinterpretations.