Modular Constraint Injection from Natural Language for CP-Based Scheduling

Florian Strohm[1], Patrick Wagner[1], Jannik Schwab[1], and Marco F. Huber[1]

Fraunhofer Institute for Manufacturing Engineering and Automation (IPA), Stuttgart, Germany
{florian.strohm, patrick.wagner, jannik.schwab, marco.huber}@ipa.fraunhofer.de

Abstract. Clinic scheduling systems built on constraint programming produce high-quality schedules, but adapting them to frequently changing operational rules remains a bottleneck: every new constraint, from staff availability changes to patient-specific restrictions, requires a developer to translate domain knowledge into solver code. We present an LLM-based agent that lets non-technical clinic staff add scheduling constraints by describing them in natural language. New constraints are injected as modular *plugins* with a standardized interface, preserving the integrity of the core scheduling model. The agent autonomously resolves entity references against the clinic database and verifies generated code through self-designed test scenarios before deployment. On a benchmark of 234 scheduling constraints across nine operationally relevant categories, the agent achieves 90.2% accuracy versus 70–82% for alternative approaches. Ablation reveals that database search and iterative self-testing are complementary, with their combination (+20.1 pp) exceeding the sum of individual contributions (+16.2 pp). The improvements hold across three LLMs, with a consistent +13–19 pp uplift over baselines.

Keywords: staff scheduling · constraint programming · large language models · agentic AI · code generation

1 Introduction

Staff scheduling in healthcare is shaped by a dense web of constraints that evolve continuously: infection-control protocols change, staff availability shifts week to week, and patients bring unique requirements [11,7]. Constraint programming (CP) has long been a natural framework for such problems [26], and modern solvers such as Google OR-Tools CP-SAT [24] routinely optimize schedules under hundreds of interacting constraints. Yet a persistent gap separates the people who *understand* scheduling rules from those who can *encode* them. Freuder [12] famously termed bridging this gap the "Holy Grail" of CP, later proposing conversational modelling as a path toward it [13]. In practice, every rule change ("Dr. Martinez cannot work Fridays," "patients over 75 should only be seen in

ground-floor rooms") must be communicated to a developer, formalized, tested, and deployed, introducing delay and miscommunication. Our work with rehabilitation clinics confirmed that even well-maintained CP systems require frequent, developer-dependent constraint updates. Large language models (LLMs) suggest a path forward: they can generate code [8,20], reason over multi-step problems [38], and use tools autonomously [39]. However, translating a natural language scheduling constraint into correct CP code requires *entity resolution* (mapping "the wheelchair patient in Building C" to a database record) and *semantic precision* (avoiding constraints that are syntactically valid but logically incorrect).

We frame our task as *incremental constraint injection*: given an already-deployed CP scheduling model and a new constraint in natural language, produce executable code that enforces the constraint *without modifying* existing logic. This differs from complete model generation [35,32]: the base model already handles core objectives, and new constraints must compose safely. Our system addresses this through (1) a *plugin architecture* injecting constraints as isolated functions with a standardized interface, and (2) a *tool-augmented LLM agent* combining database search with self-designed test scenarios.

Our contributions are as follows:

1. A **plugin-based constraint injection architecture** for CP-SAT models with modular, safe, and reversible constraint management (Sec. 3.2).
2. A **tool-augmented LLM agent** with database search and self-testing for autonomous constraint generation and verification (Sec. 3.3).
3. A **benchmark of 234 scheduling constraints** spanning nine categories with oracle-based evaluation (Sec. 4).[1]
4. An **empirical evaluation** showing 90.2% accuracy vs. 70–82% for alternatives, with complementary benefits from search and self-testing (Sec. 5).

2 Related Work

Staff scheduling and CP. Staff scheduling and rostering have been studied extensively [11,7]. CP has been a mainstay approach, with recent applications including medical appointment sequencing [2] and machine learning–CP hybrids for healthcare scheduling [4]. These efforts build or improve scheduling *algorithms*; we target the complementary problem of enabling domain experts to *customize* an existing system via natural language.

Natural language and CP. Tsouros et al. [35] frame the "Holy Grail 2.0" vision of translating natural language into constraint models, and several systems pursue it: retrieval-augmented in-context learning [21], benchmarks showing that self-verification helps [22], ReAct-style agents reaching high accuracy [32], solver

[1] Code, benchmark, and artifacts will be released upon publication.

bridges via the Model Context Protocol [31], and fine-tuning with constraint-aware retrieval [27]. Song and Cohen [29] note that *paraphrasing* the input degrades LLM accuracy, a finding directly relevant to our setting, where one constraint type recurs for many entities. Other lines embed an LLM inside the solver so propagation guides generation [5]. Adjacent efforts target sub-steps or related tasks: named-entity extraction of optimization components [10], interactive meeting scheduling [18], SAT-backed explainable scheduling [36], template-matched care-worker constraints [30], expert-guided MILP generation [19], and streamliner synthesis [37]. Most of these start from a full problem description to produce a complete formulation, target solver performance, or address only an extraction sub-step; none address *incremental constraint injection* into an already-deployed model, a task requiring entity resolution against a live database. Our work fills this gap with a plugin architecture and a self-testing loop that also improves within-pattern consistency.

Rule-based and high-level modelling languages. A long line of work lowers the barrier to changing an optimisation model without editing solver code. Domain-specific rule languages, exemplified by airline crew-scheduling systems such as Carmen [16], let planners express rostering rules in a controlled vocabulary over a fixed optimisation core; high-level constraint languages (MiniZinc [23], Essence [14], CPMpy [15], and the XCSP3 format [6]) provide declarative, solver-independent comprehensions, and the global constraint catalogue [3] standardises reusable high-level relations such as `alldifferent` and `cumulative`. These approaches are more expressive and more checkable at the modelling layer than ours: our generated constraints sit at CP-SAT's Boolean-linear level rather than using such global constructs (Section 3.4). Our contribution is orthogonal and complementary. Rather than defining a notation the user must learn and bind to correctly-spelled data, the agent resolves *informal* natural-language references against a live database at authoring time, e.g. mapping "the wheelchair patient in Building C" to rows, or matching a request for a "Spanish-speaking" staff member against the recorded `staff_languages`. A high-level modelling language could in fact serve as the agent's *generation target*, pairing schema-grounded entity resolution with compiler-checked, global-constraint output; we see this as a promising direction.

LLM-based optimization and self-debugging. The NL4Opt shared task [25] established an early benchmark of 1,101 LP problems for NL-to-optimization translation. In mathematical programming, OptiMUS [1] and others [33,40] generate optimization models from natural language. ConstraintBench [34] finds only 65% feasibility for direct LLM solutions, motivating solver-in-the-loop approaches. Iterative self-debugging [28,9] has proven effective for code generation; our agent applies similar principles with domain-specific feedback from CP-SAT feasibility.

3 System Design

3.1 Scheduling Domain

Our system assigns patient treatment needs to treatment slots in a clinic scheduling setting. A configurable *clinic generator* creates realistic synthetic instances (parameterized by staff, rooms, patients, treatment types, horizon, and target utilization) with values informed by real-world rehabilitation clinic operations. Our benchmark uses a medium-sized instance: 100 patients with conditions and allergies, 72 treatment types, 1,425 needs, 1,000 slots across 7 days, 50 staff with certifications, and 30 rooms across 5 buildings. The base CP-SAT model is deliberately minimal. Let N be the set of treatment needs and S the set of slots, and let $C \subseteq N \times S$ be the *compatibility* relation $C = \{(n, s) : \text{treatment}(s) = \text{treatment}(n)\}$ that pairs each need with the slots offering its treatment. We introduce one Boolean decision variable per compatible pair,

$$x_{n,s} \in \{0,1\}, \quad (n,s) \in C, \qquad x_{n,s} = 1 \iff \text{need } n \text{ is assigned to slot } s, \quad (1)$$

and solve

$$\begin{aligned} \max \quad & \textstyle\sum_{(n,s)\in C} x_{n,s} && && (2)\\ \text{s.t.} \quad & \textstyle\sum_{s:\,(n,s)\in C} x_{n,s} \le 1 && \forall\, n \in N, && (3)\\ & \textstyle\sum_{n:\,(n,s)\in C} x_{n,s} \le \text{cap}(s) && \forall\, s \in S. && (4) \end{aligned}$$

Constraint (3) schedules each need at most once (leaving a need unscheduled is feasible), and (4) respects each slot's capacity $\text{cap}(s)$. Under (3) the objective (2) equals the number of scheduled needs, so the solver maximizes coverage. This two-family model together with structural compatibility is the *entire* deployed base model: operational rules such as staff certification, room equipment, patient allergies, timing windows, and workload fairness are absent from it and enter only as injected plugins (Section 3.2). We fix the solver's random seed and run it single-threaded, making every solve deterministic.

3.2 Plugin Architecture

New constraints are injected as *plugins*: self-contained Python functions that extend the base model without modifying its objective or existing constraints. A plugin posts additional relations over the existing assignment variables; it never alters the objective, removes a base constraint, or rebinds an existing variable. All plugins share the fixed signature in Listing 1.1.

A plugin may introduce its own auxiliary variables (for example to count occurrences or to reify a condition), but because it only *adds* relations over the shared variables $x_{n,s}$, the set of admissible assignments can only shrink when projected onto those variables: $\pi(F') \subseteq \pi(F)$, where π projects the feasible region onto $\{x_{n,s}\}$. This yields two properties: *(i) monotone composition*, where each accepted plugin further constrains the schedule without relaxing earlier ones,

Listing 1.1. Standardized plugin interface. A plugin posts `model.Add(...)` relations over the shared `assign` variables; `context` exposes the database read-only.

```
def add_constraint(model, assign, needs_by_id, slots_by_id,
    context):
    # model:       the live CpModel being extended
    # assign:      {(need_id, slot_id): BoolVar}  -- the x_{
        n,s} above
    # needs_by_id, slots_by_id: need / slot attribute
        lookups
    # context:     read-only tables (patients, staff, rooms,
    #              certifications, allergies, equipment,
        travel times, ...)
    ...  # may add plugin-local auxiliary variables, then
        post constraints
```

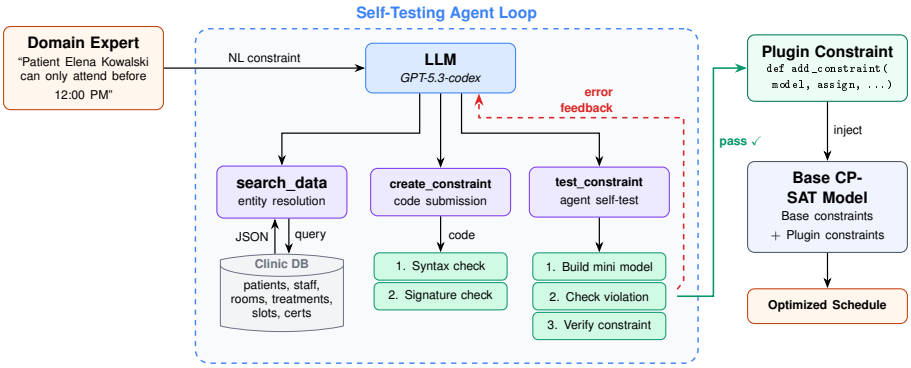

Fig. 1. System overview. A domain expert describes a constraint in natural language. The agent resolves entities via database search, generates code, verifies it through self-designed tests, and injects validated constraints as read-only plugins.

so rules compose through the shared feasible region even though their code is written independently; and *(ii) code-level isolation*, where plugins share only read-only data and the variable handles, so one plugin's code cannot corrupt the core model or another plugin. The design thus provides *modularity* (isolated constraints), *safety* (the core model cannot be overwritten), *reversibility* (constraints toggle on and off), and *auditability* (each function is independently reviewable).

3.3 Agent Architecture

Our agent (Figure 1) is a LangGraph [17] state machine iterating between an LLM and three tools for up to 10 rounds (over 95% converge within 5; the limit prevents costly unproductive loops):

`search_data(query_type, term)` searches the clinic database by entity type and returns up to 20 matching records as JSON, enabling entity resolution (e.g., "Dr. Martinez" $\rightarrow$ `staff_id 15`).

`create_constraint(code)` validates syntax and verifies that the code defines a callable `add_constraint` function.

`test_constraint(test_code)`, the key differentiator, lets the agent submit a self-designed test: a minimal CP-SAT model that checks (1) that a violation exists without the constraint and (2) that the violation is fixed with it. Tests run sandboxed with a 30-second timeout.

The system prompt provides the function signature, data schema, four CP-SAT code patterns, and seven correctness rules. In a typical loop, the agent generates code, discovers a string/integer type mismatch through `test_constraint`, fixes the comparison, and re-tests successfully.

3.4 From Natural Language to CP Rules

To make the translation concrete, consider the request *"junior-level staff members cannot be assigned to any surgical treatment."* This is not a change to a stored field: the base model contains no notion of seniority or treatment category, and no single row expresses "for every junior staff member and every surgical slot." The agent resolves `junior` against the `staff` table and `surgical` against the `treatments` table, then posts, for every compatible pair routed through a junior-staffed surgical slot, the relation

$$x_{n,s} = 0 \quad \forall\,(n,s) \in C : \mathrm{junior}\big(\mathrm{staff}(s)\big) \wedge \mathrm{surgical}\big(\mathrm{treatment}(s)\big). \tag{5}$$

The generated plugin (Listing 1.2, Appendix A) builds the `junior` and `surgical` id sets by a two-table scan, then posts Eq. (5) for every matching compatible pair. The Python loop is model *construction*, not search: it declaratively adds one linear relation per matching variable, much as a `forall` comprehension would in a higher-level modelling language. A second, aggregation-style example appears in the same appendix gallery.

4 Benchmark

4.1 Benchmark Construction

We construct a benchmark of 234 scheduling constraints designed to test a system's ability to translate diverse natural language requests into correct CP-SAT code. The constraints cover nine operationally relevant categories that reflect the rules clinic managers, head nurses, and schedulers encounter in practice: Temporal (48), Staff (38), Complex (34), Resource (31), Spatial (24), Conditional (18), Preference (17), Fairness (17), and Relational (7). For constraint types where surface-level variation is natural and frequent, such as patient-specific day restrictions ("Patient X cannot come on Mondays" vs. "Patient Y cannot

come on Thursdays") or staff daily caps, we include multiple instances sharing the same logical template but differing in entity names, parameter values, or phrasing. These *within-pattern variants* serve a dual purpose: they reflect the repetitive nature of real scheduling requests (a clinic manager will issue the same type of constraint for many different patients), and they measure *consistency*, i.e., whether a system that solves one instance of a pattern can reliably solve another. Of the 234 tests, 129 belong to 54 multi-instance pattern clusters (2–4 tests each), while 105 are unique patterns. The benchmark consists of two tiers. The first 200 constraints were initially generated using an ensemble of LLMs from a detailed prompt specifying the scheduling domain and schema. From approximately 400 candidates, we selected and refined 100 core constraints that are concrete, testable, and span all categories, then created 100 variations by rephrasing, changing parameter values, or targeting different entities. The remaining 34 constraints (tests 201–234) form a *complex tier* that stress-tests multi-step reasoning: multi-hop entity resolution, implicit/descriptive references (e.g. "the youngest surgeon"), cross-table joins over 3+ tables, compound logic with exception clauses, and tricky boundary- or domain-knowledge formulations.

4.2 Test Protocol

Each test follows a standardized three-step protocol: (1) **Baseline check**: run the scheduler *without* the constraint and verify that at least one violation exists in the resulting schedule; if no violation exists, the test is skipped as inapplicable to the current data instance. (2) **Constraint injection**: add the generated constraint to the scheduling model. (3) **Verification**: re-run the scheduler and assert that *all* assignments now satisfy the constraint oracle. This protocol ensures every test is meaningful: the constraint must actually change the schedule to pass. All 234 tests use *dynamic entity resolution*: rather than hardcoding specific patient or staff names, each test searches for entities matching required characteristics at runtime, making the benchmark *instance-independent*: the same tests work across different clinic data instances produced by the generator. Each test implements its verification oracle as a standalone Python assertion function that checks every assignment in the returned schedule, cross-referencing the scheduling database. A two-stage data pipeline (broad-coverage generation plus idempotent patching) ensures all 234 tests have baseline violations on three verified instances, enabling cross-instance robustness evaluation.

Beyond violation elimination. The protocol above is a *soundness* check: it confirms a constraint removes its target violation, but not that it leaves unrelated assignments intact; a constraint can pass while *over-restricting* the feasible region. We therefore add a *non-interference* check for exclusion-type rules: every baseline assignment that does *not* match the rule's antecedent must survive injection. The rule "patients with a life-threatening contrast-dye allergy must avoid CT-scan rooms" illustrates the gap. It affects two patients; the correct encoding excludes them from the three CT-scan rooms, but the agent's generated plugin mis-keys the equipment table (reading `equipment_name` instead of

`equipment_type`) and falls back to excluding all six imaging rooms. Both encodings satisfy the violation-elimination oracle (no affected patient occupies a CT-scan room), yet the over-broad plugin also removes both patients from the three *safe* non-CT imaging rooms, forbidding 38 additional (need, slot) pairs and scheduling one fewer need overall (objective 1421 vs. 1422). The non-interference check flags this; the soundness oracle does not. For aggregate rules, where the minimal coverage loss is itself an optimization, we instead report the objective delta as a diagnostic rather than a guarantee.

5 Experiments

5.1 Setup

We evaluate six configurations: (1) **Agent** (full system), (2) **Agent (test)** (self-testing only, no database), (3) **Agent (search)** (database only, no self-testing), and three non-agentic baselines defined below: (4) **Sampling**, (5) **Few-shot**, and (6) **Zero-shot**. All use GPT-5.3-codex (temperature 0, except Sampling), run three times for variance estimation. Each constraint is solved with CP-SAT (OR-Tools 9.15) using a single search worker and a fixed random seed (1609), making each solve deterministic; each test additionally runs under a 300-second wall-clock limit. Three data instances (seeds 1609, 2024, 7777) enable cross-instance evaluation.

Prompting baselines. The three non-agentic baselines share the agent's system prompt (function signature, data schema, four CP-SAT patterns, seven correctness rules) and emit a complete plugin in a single forward pass, with no database tool and no solver feedback. *Zero-shot* issues one generation at temperature 0 with no examples. *Few-shot* prepends five fixed, hand-written natural-language-to-code exemplars (among them a domain restriction, a linear cardinality bound, and a reified building count) and then generates once at temperature 0. *Sampling* draws N=5 independent generations at temperature 0.7 and keeps the first that passes static validation (it compiles and defines a callable `add_constraint`), a diversity strategy rather than best-of-N by quality, which is why it falls between few-shot and the agent.

5.2 Results

The agent achieves 90.2% accuracy (211/234), a 20.1 pp improvement over zero-shot (Table 1). All improvements are highly significant by McNemar's test ($p < 0.001$; Cohen's $h = 0.49$ vs. few-shot). Of 234 tests, 133 are "easy" (all pass) and 13 "hard" (all fail), leaving 88 discriminating tests where the agent achieves 88.6% vs. 62.5–67.0% for the next-best systems.

Table 1. Overall results (n=234) and discriminating tests (88 tests where systems differ).

System	Accuracy	Discrim. (n=88)
Agent	**90.2%**	**88.6%**
Agent (test)	82.1%	67.0%
Sampling (N=5)	80.3%	62.5%
Agent (search)	74.4%	46.6%
Few-shot	71.4%	38.6%
Zero-shot	70.1%	35.2%

Table 2. Accuracy (%) by category (selected); category size n in parentheses.

	Temporal (n=48)	*Complex* (n=34)	*Staff* (n=38)	*Resource* (n=31)	*Conditional* (n=18)	*Fairness* (n=17)
Agent	**93.8**	**91.2**	**81.6**	**93.5**	**100.0**	**76.5**
Agent (test)	85.4	58.8	**81.6**	87.1	72.2	82.4
Sampling	83.3	55.9	76.3	90.3	83.3	82.4
Zero-shot	77.1	52.9	73.7	74.2	72.2	41.2

Ablation: search and self-testing are complementary. Measured against zero-shot, self-testing alone contributes +12.0 pp (82.1%) and database search alone +4.3 pp (74.4%). Yet each mechanism is far more effective in the presence of the other: self-testing adds +15.8 pp on top of search (74.4→90.2%), and search adds +8.1 pp on top of self-testing (82.1→90.2%). The combined +20.1 pp therefore exceeds the sum of the two standalone contributions (+16.2 pp), indicating that the mechanisms reinforce each other.

Category analysis. Our per-category breakdown (Table 2; selected categories) shows where capabilities matter most. *Complex constraints* (91.2% vs. 53–59%) show the starkest separation: these 34 tests require multi-hop entity resolution and compound logic, and systems without database search perform far worse. *Conditional constraints* reach 100% (vs. 72–89%) by combining search with self-testing. *Temporal* and *resource* constraints (93.5–93.8%) benefit from self-testing catching type errors and infeasibility. The search-only agent underperforms few-shot on several categories (e.g., Temporal: 70.8% vs. 85.4%), suggesting database results can introduce distracting context when the agent lacks execution feedback; self-testing is needed to realize the benefit of entity resolution.

Error analysis. Of 23 non-passing tests, 7 are solver timeouts (all fairness constraints with $O(|\text{staff}| \times |\text{days}|)$ aggregation) and 16 logic errors. Root causes:

missing cross-table joins (5/16), incorrect value comparisons (5/16), wrong entity resolution (3/16), and constraints that are semantically too weak, passing self-tests but not fully capturing user intent (3/16). The last category is clinically concerning: a constraint intended to *exclusively* assign morning slots might merely add a soft *preference*, a semantic gap that our self-testing cannot detect.

Robustness and generalization. The agent is stable across runs (90.2±0.9%; 218/234 tests identical), data instances (89.3±0.9% across three seeds), and model families: GPT-5.1-Mini and Kimi-K2.5 (open-weights) show a consistent +13–19 pp uplift over baselines. It also generalizes within constraint *patterns*: across the 54 multi-instance clusters it solves all instances of a pattern consistently in 87.0% of cases (47/54) versus 72.2% (39/54) for zero-shot. Overhead is moderate: it passes zero-shot within 60 s per constraint and sampling at 120 s, with diminishing returns beyond 180 s.

6 Discussion

Why the agent loop works. The agent's 20.1 pp improvement over zero-shot generation stems from two complementary error-correction mechanisms, each targeting a different failure mode.

Self-testing catches code-level errors. A failed self-test pinpoints the fault (a type mismatch, off-by-one error, or missing edge case), which the agent fixes iteratively, usually within 2–3 rounds. The benefit grows with complexity: complex constraints (tests 201–234) combine independently-failing sub-conditions, explaining the agent's 91.2% there versus 53–59% for single-pass systems that must get every sub-condition right at once.

Database search resolves data-level errors. Constraints reference entities, field values, and cross-table relationships absent from the constraint text, so without data access the LLM must guess, e.g. testing `insurance_type == "Public"` when the database stores `"public"`, silently constraining no one. The agent instead queries the database for actual values before generating code.

Practical implications. The plugin architecture matters for deployed scheduling tools: where constraints span staff qualifications, equipment, and patient conditions, adding them without risking the core logic is operationally important, and different stakeholders can add rules independently with reversibility and audit trails. For clinical deployment we recommend three safeguards: a *dry-run mode* showing a before/after schedule diff so experts can validate effects without reading code; audit trails with rollback; and human approval before activation (decision support, not automation).

Limitations. The self-testing loop verifies that the augmented model remains *feasible* but does not guarantee *semantic correctness*: a constraint that is too weak (e.g., restricting only Monday instead of all weekdays) may pass the feasibility test while not capturing the user's intent. Three of 16 logic errors fall into this

"too weak constraint" category. Conversely, an overly strict constraint can cause harm by excluding valid assignments, e.g., denying a patient necessary treatment slots. Because each plugin can only shrink the feasible region projected onto the assignment variables (Section 3.2), violation-elimination alone cannot detect this, and the agent's own self-test shares the blind spot. The benchmark-side non-interference check of Section 4.2 catches such over-restriction for exclusion-type rules, but an in-loop minimality check for the deployed agent (and coverage auditing for aggregate constraints, where minimality is itself an optimization) remains future work. We evaluate on clinic scheduling only; generalization to other scheduling domains (nurse rostering, university timetabling, job-shop scheduling) requires further study, though the plugin interface pattern is domain-agnostic in principle. The scheduling data is synthetically generated with clean naming; real-world data may introduce messier entity names, abbreviations, and missing fields. Finally, our benchmark is single-turn; although our live demonstrator supports multi-turn clarification (e.g., disambiguating a shared surname), neither this nor an end-to-end user study with clinic staff is yet formally evaluated.

7 Conclusion

We presented an LLM-based agent for translating natural language scheduling constraints into executable CP-SAT code, making deployed scheduling systems accessible to non-technical domain experts. Our plugin architecture ensures that generated constraints compose safely with existing scheduling logic, providing modularity, safety, reversibility, and auditability, properties essential for operational scheduling environments. Evaluation on 234 scheduling constraints spanning nine operationally relevant categories yields two principal findings. First, the agent achieves 90.2% accuracy versus 70–82% for five alternative approaches, with the gap widening to 88.6% versus 35–67% on 88 discriminating tests ($p < 0.001$ by McNemar's test). The advantage is most pronounced on 34 complex tests requiring multi-hop reasoning: 91.2% versus 53–59% for baselines. Second, ablation shows self-testing and data access are complementary: together they yield +20.1 pp, exceeding the sum of their standalone gains. Beyond the agent itself, our benchmark of 234 constraints with oracle-based evaluation provides a reusable testbed for future work on LLM-driven constraint generation in scheduling. Cross-model evaluation on three LLMs, including an open-weights model, confirms that the benefit generalizes across model families (+13–19 pp uplift). Future work includes extending the plugin architecture to other scheduling domains (nurse rostering, university timetabling, job-shop scheduling); supporting constraint *modification* and *retraction*, since the current system only adds constraints; lifting generated constraints to higher-level global-constraint forms; and adding multi-turn user interaction and soft-constraint generation, directions the plugin interface can accommodate but that we have not yet implemented or benchmarked.

Use of AI Assistance. The constraint-generation agent studied here is itself LLM-based (GPT-5.3-codex, with GPT-5.1-Mini and Kimi-K2.5 in the cross-model experiments). Large language models also assisted the authors: the benchmark's natural-language constraints were proposed by an LLM ensemble and selected by the authors, and the verification oracles and experiment code were implemented collaboratively through agentic coding under author review; Claude (Anthropic) additionally supported research discussions and manuscript editing. No experimental result, table entry, or statistical test was produced by an LLM; all reported numbers come from the deterministic OR-Tools harness run by the authors, who take full responsibility for the paper.

References

1. AhmadiTeshnizi, A., Gao, W., Udell, M.: OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In: Proc. ICML (2024)
2. Assaf, G., Löffler, S., Hofstedt, P.: Constraint-based optimization for scheduling medical appointments. In: Proc. ICAART. vol. 3, pp. 94–103. SCITEPRESS (2025)
3. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalogue: Past, present and future. Constraints **12**(1), 21–62 (2007)
4. Ben Said, A., Mouhoub, M.: Machine learning and constraint programming for efficient healthcare scheduling. arXiv preprint arXiv:2409.07547 (2024)
5. Bonlarron, A., Régin, F., De Maria, E., Régin, J.C.: Large language model meets constraint propagation. In: Proc. IJCAI. pp. 10036–10044 (2025)
6. Boussemart, F., Lecoutre, C., Piette, C.: XCSP3: An integrated format for benchmarking combinatorial constrained problems. arXiv preprint arXiv:1611.03398 (2016)
7. Burke, E.K., De Causmaecker, P., Vanden Berghe, G., Van Landeghem, H.: The state of the art of nurse rostering. J. Scheduling **7**(6), 441–499 (2004)
8. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
9. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. In: Proc. ICLR (2024)
10. Dakle, P.P., Kadioğlu, S., Uppuluri, K., Politi, R., Raghavan, P., Rallabandi, S., Srinivasamurthy, R.: Ner4Opt: Named entity recognition for optimization modelling from natural language. In: Proc. CPAIOR. pp. 299–319. Springer (2023)
11. Ernst, A.T., Jiang, H., Krishnamoorthy, M., Sier, D.: Staff scheduling and rostering: A review of applications, methods and models. Eur. J. Oper. Res. **153**(1), 3–27 (2004)
12. Freuder, E.C.: In pursuit of the holy grail. Constraints **2**(1), 57–61 (1997)
13. Freuder, E.C.: Conversational modeling for constraint satisfaction. In: Proc. AAAI. vol. 38, pp. 22592–22597 (2024)
14. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
15. Guns, T.: Increasing modeling language convenience with a universal n-dimensional array: CPpy as a python-embedded example. In: Proc. 18th Workshop on Constraint Modelling and Reformulation (ModRef) at CP (2019)

16. Kohl, N., Karisch, S.E.: Airline crew rostering: Problem types, modeling, and optimization. Annals of Operations Research **127**, 223–257 (2004)
17. LangChain: LangGraph: Building stateful, multi-actor applications with LLMs (2024)
18. Lawless, C., Schoeffer, J., Le, L., Rowan, K., Sen, S., St. Hill, C., Suh, J., Sarrafzadeh, B.: I want it that way: Enabling interactive decision support using large language models and constraint programming. ACM Trans. Interact. Intell. Syst. **14**(3), 1–32 (2024)
19. Li, Q., Zhang, L., Mak-Hau, V.: An LLM-powered MILP modelling engine for workforce scheduling guided by expert knowledge. arXiv preprint arXiv:2511.02364 (2025)
20. Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al.: StarCoder: May the source be with you! In: arXiv preprint arXiv:2305.06161 (2023)
21. Michailidis, K., Tsouros, D., Guns, T.: Constraint modelling with LLMs using in-context learning. In: Proc. CP (2024)
22. Michailidis, K., Tsouros, D., Guns, T.: DCP-Bench-Open: Evaluating LLMs for constraint modelling of discrete combinatorial problems. arXiv preprint arXiv:2506.06052 (2025)
23. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Proc. Principles and Practice of Constraint Programming (CP). LNCS, vol. 4741, pp. 529–543. Springer (2007)
24. Perron, L., Furnon, V.: OR-Tools. In: Google Operations Research Tools (2023)
25. Ramamonjison, R., Yu, T., Li, R., et al.: NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In: Proc. NeurIPS Competitions Track, PMLR. vol. 220, pp. 189–203 (2023)
26. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2. Elsevier (2006)
27. Shi, W., Liu, M., Zhang, W., Shi, L., Jia, F., Ma, F., Zhang, J.: ConstraintLLM: A neuro-symbolic framework for industrial-level constraint programming. In: Proc. EMNLP. pp. 15999–16019 (2025)
28. Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: Language agents with verbal reinforcement learning. In: Proc. NeurIPS. vol. 36 (2023)
29. Song, Y., Cohen, E.: Do LLMs understand constraint programming? Zero-Shot constraint programming model generation using LLMs. In: Proc. LION. Lecture Notes in Computer Science, vol. 15744. Springer (2025)
30. Suenaga, K., Furuta, T., Ono, S.: A study on constraint extraction and exception exclusion in care worker scheduling. Artif. Life Robot. (2025)
31. Szeider, S.: Bridging language models and symbolic solvers via the model context protocol. In: Proc. SAT. pp. 30:1–30:12 (2025), arXiv:2501.00539
32. Szeider, S.: CP-Agent: Agentic constraint programming. arXiv preprint arXiv:2508.07468 (2025), to appear at AGENT@ICSE 2026
33. Thind, R., Sun, Y., Liang, L., Yang, H.: OptimAI: Optimization from natural language using LLM-powered AI agents. arXiv preprint arXiv:2504.16918 (2025)
34. Tso, J., Schmittou, P., Huynh, Q., Hutchins, J.: ConstraintBench: Benchmarking LLM constraint reasoning on direct optimization. arXiv preprint arXiv:2602.22465 (2026)
35. Tsouros, D., Verhaeghe, H., Kadioğlu, S., Guns, T.: Holy grail 2.0: From natural language to constraint models. arXiv preprint arXiv:2308.01589 (2023)
36. Vasileiou, Y., Yeoh, W.: TRACE-CS: A synergistic approach to explainable course scheduling using LLMs and logic. In: Proc. AAAI (2025)

37. Voboril, F., Peruvemba Ramaswamy, V., Szeider, S.: StreamLLM: Enhancing constraint programming with large language model-generated streamliners. In: Proc. NSE@ICSE (2025), arXiv:2408.10268
38. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. Proc. NeurIPS **35**, 24824–24837 (2022)
39. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: ReAct: Synergizing reasoning and acting in language models. In: Proc. ICLR (2023)
40. Zhang, B., Luo, P., Yang, G., Soong, B.H., Yuen, C.: OR-LLM-Agent: Automating modeling and solving of operations research optimization problems with reasoning LLM. arXiv preprint arXiv:2503.10009 (2025)

A Generated Constraint Gallery

We give two generated plugins verbatim (apart from removed comments), each posting only `model.Add(...)` relations over the shared `assign` variables of Section 3.1: the worked example of Section 3.4 (Listing 1.2) and a second, aggregation-style constraint (test 117).

Listing 1.2. Generated plugin for "junior staff cannot do surgical treatments" (test 39, the worked example of Section 3.4).

```
def add_constraint(model, assign, needs_by_id, slots_by_id,
    context):
    junior = {sid for sid, s in context["staff"].items()
              if str(s.get("seniority_level","")).lower() ==
                  "junior"}
    surgical = {tid for tid, t in context["treatments"].
        items()
                if str(t.get("treatment_category","")).lower
                    () == "surgical"}
    for n in context["need_ids"]:
        for s in context["compatible_slots"].get(n, []):
            slot = slots_by_id.get(s)
            if slot and slot["staff_id"] in junior and slot[
                "treatment_id"] in surgical:
                model.Add(assign[(n, s)] == 0)
```

Aggregation (test 117): "each building should handle at most 8 appointments per day." The plugin groups assignment variables by the `(building, day)` key (an aggregation the base model never computes) and posts one Boolean-linear at-most-8 inequality per group.

```
def add_constraint(model, assign, needs_by_id, slots_by_id,
    context):
    from collections import defaultdict
    rooms = context["rooms"]
    by_bd = defaultdict(list)
    for n in context["need_ids"]:
```

```
        for s in context["compatible_slots"].get(n, []):
            slot = slots_by_id.get(s)
            room = rooms.get(slot["room_id"]) if slot else
                None
            if room and (n, s) in assign:
                by_bd[(room["building"], slot["day_of_week"
                    ])].append(assign[(n, s)])
    for vars_list in by_bd.values():
        model.Add(sum(vars_list) <= 8)
```