

SAGenT: From Natural Language to SAT with LLM Agents

Guillaume Atencia
Carnegie Mellon University

Ruben Martins
Carnegie Mellon University

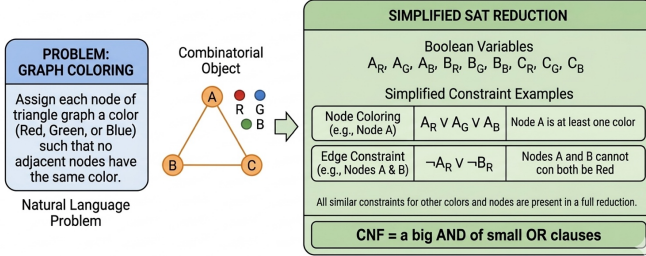


Figure 1: Simplified example of reducing a natural-language graph coloring problem into Boolean variables and CNF constraints.

1 Introduction

Modern SAT solvers are powerful tools for solving combinatorial problems such as scheduling, graph coloring, routing, allocation, and puzzles. Once a problem is correctly encoded as Boolean satisfiability, the solver can often search the solution space very efficiently. The hard part is usually the modeling step: translating a natural-language problem into Boolean variables, constraints, CNF clauses, pseudo-Boolean constraints, or another SAT-compatible representation. This requires both understanding the problem and knowing how to express it formally [1, 2].

SAGenT studies whether large language models can reduce this modeling burden by acting as SAT modeling agents rather than direct problem solvers. The central question is whether an LLM-guided agent can take a natural-language constraint problem, construct a correct SAT-style encoding, call a symbolic solver, and decode the result back into the original problem domain. The goal is not to replace SAT solvers, but to build an agentic front end that makes them easier to use.

The system follows the main stages of SAT modeling: identify variables, encode constraints, choose a backend, call the solver, and decode the Boolean assignment. It supports CNF, pseudo-Boolean / PySAT-style encodings, and an experimental MiniZinc backend. A key design choice is to rely on existing encoding libraries where possible, especially PySAT and pseudo-Boolean encoders, so the LLM does not need to manually expand every high-level constraint into raw CNF [3, 4].

The project begins with a ReAct-style baseline agent, where the model iteratively reasons, acts through the SAT system, observes feedback, and revises its encoding [5]. The later architecture extends this into a GALA-inspired multi-candidate system. Instead of relying on one modeling attempt, it generates several

candidate encodings through specialized proposal components. These candidates explore different variable schemas, constraint decompositions, and backend strategies. A verifier-gated ranking pipeline then checks structural sanity, grounding, decoder compatibility, fuzzing behavior, and solver feedback before selecting an encoding [7].

The main contributions are fivefold. First, I built an end-to-end LLM-to-SAT pipeline for modeling, solving, and decoding natural-language constraint problems. Second, I compared intermediate representations between natural language and SAT, especially PySAT / pseudo-Boolean encodings versus MiniZinc. Third, I implemented a GALA-inspired proposal architecture to explore multiple encodings instead of depending on one generation. Fourth, I added verifier-gated ranking and solver feedback to reduce incorrect encodings. Finally, I evaluated the system on benchmark combinatorial problems to study where agentic SAT modeling succeeds, where it fails, and which representations are most practical.

2 Background and Related Work

This project fits into natural-language-to-formal-model translation: turning an informal problem description into a precise model that a symbolic solver can execute. SAT is an important target because many combinatorial problems reduce to Boolean satisfiability. However, users must still choose variables, constraints, and encodings. SAGenT focuses on this modeling bottleneck.

One important comparison is CP-Agent, which treats constraint modeling as an agentic coding problem rather than a fixed prompting task. CP-Agent uses a ReAct-style Python coding agent with file operations, code execution tools, and a persistent IPython kernel. This lets the agent iteratively build, test, and debug constraint models. In the constraint-programming setting, it uses CPMpy and a project prompt containing domain-specific modeling guidance. CP-Agent reports solving all 101 CP-Bench problems, arguing that success comes from testing hypotheses, using execution feedback, and adapting the workflow [6, 8].

SAGenT adopts the same general lesson: natural-language modeling is not just text generation. It often requires the kind of loop a human modeler would use: write a candidate model, run it, inspect failures, and revise. The difference is that SAGenT targets SAT-compatible encodings such as CNF and pseudo-Boolean constraints rather than CPMpy-style constraint programs.

A second influence is GALA, a multi-agent framework for text-to-model translation in MiniZinc. GALA de-

composes modeling across specialized agents focused on global constraint types. Each agent detects whether a constraint such as `all_different`, `cumulative`, or `global_cardinality` appears, and an assembler combines the snippets into a complete MiniZinc model. GALA reports improvements over direct prompting and chain-of-thought baselines, suggesting that decomposing the modeling problem can improve reliability [7, 9].

SAgent adapts this idea to SAT. Instead of only detecting MiniZinc global constraints, its GALA-style components explore alternative SAT encodings, variable schemas, and backend choices. The goal is to avoid overcommitting to a single modeling direction too early.

Intermediate representation choice is also central. MiniZinc and other constraint-programming languages are high-level and declarative, but they introduce another compiler layer and may reduce direct control over the final SAT encoding. Pseudo-Boolean and MaxSAT representations are closer to SAT while still compactly expressing sums, cardinalities, and weighted objectives. SMT is more expressive for arithmetic, arrays, bit-vectors, and typed formulas, but relies on SMT solver internals. ASP is natural for rule-based discrete problems, but requires a separate grounding and solving ecosystem. Direct CNF gives maximum control, but is the most brittle target for an LLM because every high-level constraint must be manually decomposed into clauses [10, 11].

SAgent focuses on PySAT and pseudo-Boolean encodings because they occupy a useful middle ground. They are close enough to SAT to preserve solver compatibility, but expressive enough that the LLM does not need to hand-write every clause. This is especially useful for cardinality and bounded-sum constraints, where pseudo-Boolean APIs can perform the CNF expansion reliably [3, 2].

3 Problem Setting and Research Goal

The task is to translate a natural-language combinatorial problem into a correct SAT-compatible model. Given an input problem, the system must identify relevant objects, define Boolean variables or variable families, generate constraints, call a SAT/PySAT-style backend, and decode the satisfying assignment into a domain-level solution. The output is not just “SAT” or “UNSAT,” but a coloring, schedule, path, placement, assignment, or other structured answer.

The central challenge is that the system is searching for the right model, not just the answer. It must decide which entities matter, how they should be represented, which constraints are required, whether the task is satisfaction or optimization, and how conditions such as “exactly one,” “at most one,” reachability, movement, or capacity should be encoded. It also needs a decoding map so that a Boolean assignment becomes an interpretable solution.

A candidate encoding can be syntactically valid and still be wrong. It may omit a rule, model the wrong objects, use ungrounded variables, allow illegal assignments, or produce a

solver result that cannot be decoded. For example, an encoding might allow two objects to occupy the same slot even though the problem forbids it, or it might solve a relaxed version of the task because a cardinality constraint was translated incorrectly. In these cases, the solver may return SAT, but only for the flawed model.

SAgent is designed around these failure modes. The LLM may choose the wrong modeling strategy, miss constraints, introduce meaningless variables, generate a semantically wrong formula, produce an undecodable assignment, or create a formula that is too large. The research goal is to test whether a structured architecture can reduce these failures. The project compares a ReAct baseline, where one agent iteratively builds an encoding, with a GALA-inspired architecture that generates multiple encodings, verifies them, ranks them, and selects the most promising candidate. The main hypothesis is that SAT modeling benefits from exploration and verification instead of trusting a single LLM-generated model.

4 System Design: SAgent Architecture

SAgent is an LLM-guided SAT modeling system that moves a natural-language problem through specification extraction, candidate generation, compilation, verification, solving, and decoding. The LLM proposes formal encodings, while the symbolic solver checks satisfiability once a candidate formula has been produced.

The input is a problem statement. The first stage extracts entities, dimensions, constraints, and expected output structure. This specification guides candidate encodings, each of which includes a variable schema, constraints, backend target, and decoding information. The core pipeline is: define variables, encode constraints, compile to a backend artifact, solve, and decode.

The baseline uses a ReActAgent loop. The LLM alternates between reasoning and acting through the SAT system. It can define variables, add constraints, test partial encodings, call the solver, and revise based on feedback. This baseline tests whether a single iterative trajectory can build a useful encoding [5].

SATManager serves as the main orchestration layer. Conceptually, it is the control tower: the LLM chooses modeling actions, while SATManager turns those actions into structured state and eventually into a solver artifact. The broader engine includes modules for the agent loop, orchestration, proposal generation, ranking, backends, compilation, solving, and telemetry.

Variable management is separated because SAT encodings require stable allocation. The system maps domain objects such as nodes, colors, robots, time steps, tasks, or grid cells into Boolean variables. This mapping must be deterministic so that constraints, rankings, and decoding remain comparable, especially when several candidates use different schemas.

Constraint creation and compilation are also separated. The agent does not simply output raw DIMACS; it builds constraints through structured actions and intermediate representations. These are compiled into backend-specific artifacts, allow-

ReAct SAT modelling pipeline

Solve mode: solve, decode, finish
Encoding mode: verify, auto-submit candidate

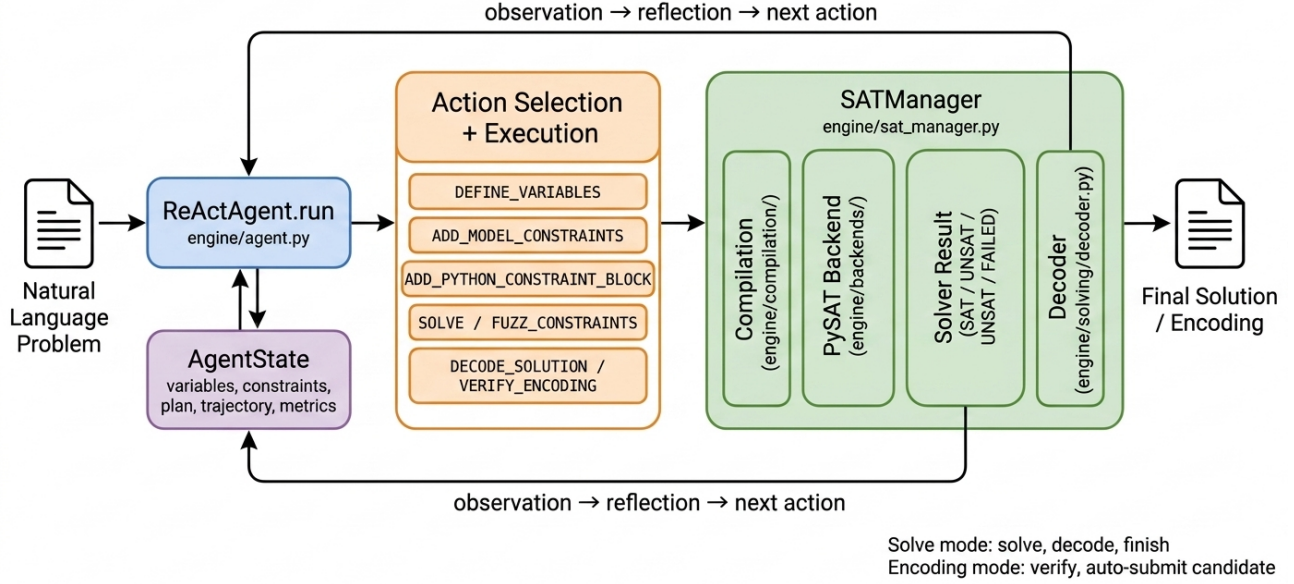


Figure 2: ReAct SAT modeling pipeline. The agent repeatedly observes, reflects, and chooses actions such as defining variables, adding constraints, solving, and decoding.

ing the system to compare CNF, pseudo-Boolean, and MiniZinc-style routes without rewriting the whole agent loop.

The most practical backend in the current design is the PySAT / pseudo-Boolean path. Many problem statements contain constraints such as “exactly one,” “at most one,” or “sum below a bound.” These are natural pseudo-Boolean constraints, and existing encoders can compile them into CNF. This avoids forcing the LLM to manually write large numbers of low-level clauses [3, 2].

After solving, the decoder maps the Boolean assignment back into the original problem format. For example, it converts true variables into node colors, robot paths, task times, or piece placements. This is critical because a SAT result is only useful if it can be interpreted correctly. A mismatch between the variable schema and decoder can make a technically satisfiable result unusable.

Telemetry records run behavior, failures, solver outcomes, and benchmark results. It helps identify where a run failed: specification extraction, grounding, constraint generation, compilation, solving, ranking, or decoding.

The GALA-inspired extension sits on top of the ReAct baseline. Instead of trusting one trajectory, SAgent generates multiple candidates and ranks them before solving. Candidates may differ in variable schema, constraint decomposition, backend choice, cardinality encoding, symmetry breaking, auxiliary variables, and decoder assumptions. This turns modeling into a search over reductions rather than a one-shot generation problem.

Denabase is a planned or partially integrated memory layer. It is intended to store prior SAT encodings, structural fingerprints,

gadget patterns, and solver traces, then retrieve structurally similar examples for future problems. In this report, it should be treated as a future-oriented memory component rather than the main evaluated mechanism.

5 Intermediate Representations and Encoding Choices

A central design question is what representation the LLM should generate before the final SAT solver input. Every problem must eventually become SAT-compatible, but the LLM does not need to write raw DIMACS CNF directly. It can target an IR that is easier to generate and inspect while still compiling reliably to SAT.

The project considered several IR families. MiniZinc and other constraint-programming languages are attractive because they are high-level and declarative. They express decision variables, domains, arithmetic relations, and global constraints in a compact way. For example, graph coloring can be modeled with one color variable per node and constraints that adjacent nodes differ. However, MiniZinc adds a compiler and backend layer between the LLM and SAT. In this implementation, that made it more fragile than the PySAT route [4].

Pseudo-Boolean constraints and PySAT-style encodings were the most useful middle ground. A pseudo-Boolean constraint is a linear inequality over Boolean variables, such as a sum bounded by k . Many common constraints fit this form: exactly one color per node, at most one task per time slot, or selected items under a budget. Existing encoders can compile

these constraints into CNF, reducing the chance of LLM clause-generation errors [2, 3].

SMT and ASP were considered but not used as the main backend. SMT is expressive for typed variables, arithmetic, arrays, and bit-vectors, but relies on SMT solver internals rather than a direct SAT/PySAT pipeline. ASP is expressive for discrete “guess-and-check” problems, but requires a separate grounding and solving system. Both are promising future directions [10, 11].

Direct CNF gives the most control because every clause is explicit, but it is the least forgiving target. Even simple ideas can require many clauses. For example, “exactly one” requires both an at-least-one clause and an at-most-one encoding. Asking an LLM to repeatedly generate these low-level encodings increases the risk of subtle logical errors.

The main design finding is that PySAT / pseudo-Boolean encodings were the most practical IR for this system. They balanced expressiveness, solver compatibility, and implementation effort. MiniZinc remained conceptually attractive, but introduced more integration fragility. Raw CNF gave control, but was too brittle for LLM generation. The most reliable path was therefore the middle representation: pseudo-Boolean constraints compiled through a PySAT-style backend.

6 GALA-Inspired Multi-Agent Proposal Generation

The ReAct baseline gives the LLM an iterative tool-using loop, but it has an early-commitment problem. Once the agent chooses a bad variable schema or encoding direction, later steps often refine the same flawed model instead of replacing it. Solver feedback may show that something is wrong, but it does not always cause the agent to rethink the representation.

The GALA-inspired extension addresses this by generating multiple candidate encodings before solving. Different proposal slots or specialist-style agents explore different modeling strategies. The system asks not whether one agent can repair its first idea, but whether several plausible encodings can be generated and then filtered.

The proposal generator receives the extracted specification and produces multiple IR candidates. Each candidate contains a variable schema, constraint structure, backend choice, and decoding assumptions. The candidates are meant to be genuinely different modeling hypotheses, not superficial rewrites.

Diversity is encouraged through specialist slots and schema families. One slot may prefer direct encodings, where variables correspond to choices such as “node v has color c ” or “robot r is at cell x at time t .” Another may explore order-style encodings, where variables represent comparisons, prefixes, or thresholds. A log-style encoding may represent larger domains more compactly. Slot contracts define expectations for each proposal type and reduce the chance that all agents collapse to the same default encoding.

After proposal generation, an assembler or normalization step

turns the outputs into comparable IR candidates. This lets later stages compile, verify, rank, and solve them consistently.

The key point is that the agents explore different models, not just different phrasings. In graph coloring, one candidate might use direct node-color variables, while another might explore a different cardinality or compact color representation. In scheduling, one candidate might use task-time variables, while another uses ordering variables. These are different reductions, so failure in one does not imply failure in all.

This matters because many SAT failures are representation failures. A good variable schema can make the constraints and decoder simple; a bad schema can make the entire pipeline brittle. Multi-candidate generation increases the chance that at least one proposal lands in the right modeling family, while the verifier and ranking stages decide which candidate to trust.

7 Verification, Ranking, and Solver Pipeline

SAgentT cannot trust LLM-generated encodings directly. A candidate may compile and even return SAT while still representing the wrong problem. The verifier-gated ranking pipeline filters invalid candidates, assigns confidence to plausible ones, and selects an encoding for solving.

Stage 0 performs structural sanity checks. It catches obvious issues such as missing variables, missing constraints, implausible backend choices, or empty proposals. This stage does not prove correctness, but it avoids wasting time on malformed candidates.

Semantic verification checks whether the encoding behaves like the intended problem rather than merely compiling. This includes SAT-oracle-style signals that help detect hollow, over-relaxed, or over-constrained encodings. A useful candidate should enforce the rules it claims to model.

Grounding validation checks whether variables used in constraints are declared, meaningful, and tied to the input instance. LLMs can introduce plausible-sounding variables that are not connected to real problem entities. Grounding asks whether the model’s variables actually correspond to the problem.

Decoder compatibility checks whether the variable schema can be read back into a solution. Since a SAT solver returns only a Boolean assignment, the decoder must know how to interpret true variables as a graph coloring, schedule, path, or placement. If the schema and decoder disagree, the result may be unusable.

Fuzzing and micro-instance checks test the encoding on small controlled cases. Some cases should be satisfiable; others intentionally violate a constraint and should be unsatisfiable. These tests catch modeling bugs that syntax checks miss.

Formula statistics, such as variable count, clause count, clause width, and structural patterns, provide additional signals. They do not prove correctness, but they can identify suspiciously empty, bloated, or inefficient encodings.

The ranking stage combines these signals into a hand-designed score. The goal is practical trust, not a proof of correctness.

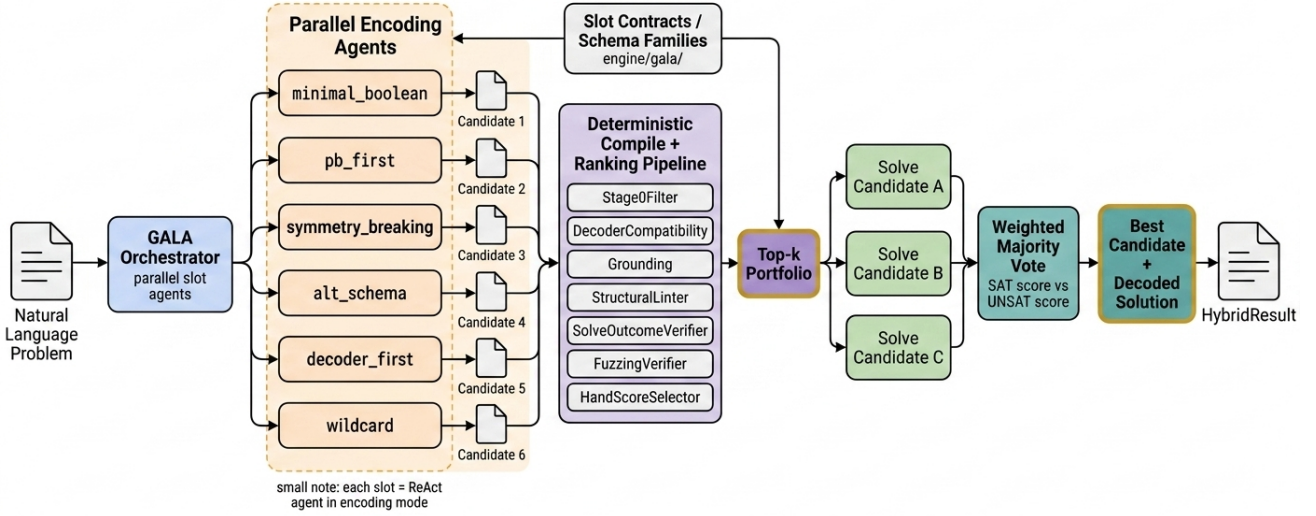


Figure 3: GALA-inspired multi-candidate architecture. Parallel encoding agents generate different candidates, which are compiled, verified, ranked, and solved as a top-k portfolio.

Table 1: Main ranking and verification signals.

Signal	Purpose
Stage 0	Filters malformed or empty candidates.
Semantic verification	Tests whether the encoding behaves like the intended problem.
Grounding	Checks whether variables are tied to problem entities.
Decoder compatibility	Checks whether SAT assignments can be read back into solutions.
Fuzzing / micro-tests	Tests small SAT/UNSAT cases.
Formula statistics	Flags suspiciously empty, large, or inefficient encodings.

A candidate with strong grounding, compatible decoding, good semantic checks, successful fuzzing, and reasonable formula statistics should rank above one that only compiles.

After ranking, selected candidates are compiled into CNF or pseudo-Boolean/PySAT-compatible artifacts. The backend calls the SAT solver and returns a satisfiability result and, when SAT, a Boolean assignment. The decoder maps this assignment back into the original problem format. If the top-ranked candidate fails during compilation, solving, or decoding, the system can fall back to another candidate.

Determinism matters throughout this process. Candidate comparison only makes sense if variable allocation, constraint IDs, and compilation order are stable. Otherwise, rankings and de-

bugging become noisy. Verification and ranking therefore bridge creative LLM proposal generation and reliable symbolic solving.

8 Experimental Setup

The system was evaluated on benchmark natural-language combinatorial problems. The benchmark suite included graph coloring, Hamiltonian cycle, jobshop scheduling, multi-robot path planning, number partitioning, polyomino exact cover / pentomino, Slitherlink, and Sokoban. These families cover assignment constraints, graph structure, temporal reasoning, paths, packing, exact cover, and puzzle-style local consistency.

Each instance is passed to the agent as a natural-language or structured input. The agent generates an encoding, compiles it, calls the solver, decodes the assignment, and submits the decoded answer to a family-specific checker. This matters because solver success alone is not enough: a result only counts if the decoded solution satisfies the original problem rules.

The main comparison is between the ReAct baseline and the GALA-inspired architecture. ReAct uses one iterative LLM trajectory. GALA generates multiple candidates and uses verification/ranking before solving. This tests whether exploring several encodings is more reliable than relying on one trajectory.

The project also included ablations such as full GALA, fewer slots, no alternative schema, no wildcard, no symmetry, semantic ranking off, decoder compatibility off, top-1 only, top-2 fallback, and IR enforcement on/off. These isolate the value of proposal diversity, schema variation, symmetry guidance, semantic

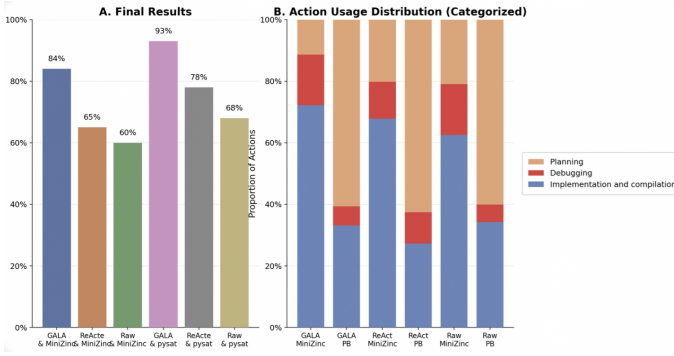


Figure 4: Final pass rates and categorized action usage. The strongest configuration was GALA with PySAT / pseudo-Boolean encodings, which reached 93% pass rate.

Table 2: Final pass rates by architecture and intermediate representation.

Architecture	IR	Pass rate
GALA-inspired	PySAT / PB	93%
GALA-inspired	MiniZinc	84%
ReAct baseline	PySAT / PB	78%
ReAct baseline	MiniZinc	65%
Raw / prompt-based	PySAT / PB	68%
Raw / prompt-based	MiniZinc	60%

ranking, decoder checks, fallback behavior, and IR enforcement.

The main metric is pass rate: the percentage of runs where the decoded solution is accepted by the benchmark checker. The experiments also track solver success/failure, per-family success, and runtime. Per-family results are important because each benchmark stresses different modeling skills: graph families stress graph encodings, jobshop and MRPP stress temporal structure, number partitioning stresses arithmetic/cardinality, and Slitherlink or Sokoban stress puzzle-specific constraints and decoding.

9 Results

The evaluation shows two main results. First, the GALA-inspired architecture improves success rate over the ReAct and raw baselines. Second, the choice of intermediate representation has a large effect on both success rate and agent behavior: PySAT / pseudo-Boolean encodings were the most practical representation in this implementation.

The clearest result is the final success-rate comparison. The best-performing configuration is GALA with PySAT / pseudo-Boolean encodings, which reaches 93% pass rate. The next best configuration is GALA with MiniZinc, at 84%. ReAct with PySAT reaches 78%, while ReAct with MiniZinc reaches 65%. The raw baselines are lower: 68% with PySAT and 60% with MiniZinc. This ranking shows that both design choices matter. GALA improves over ReAct, and PySAT / pseudo-Boolean encodings improve over MiniZinc.

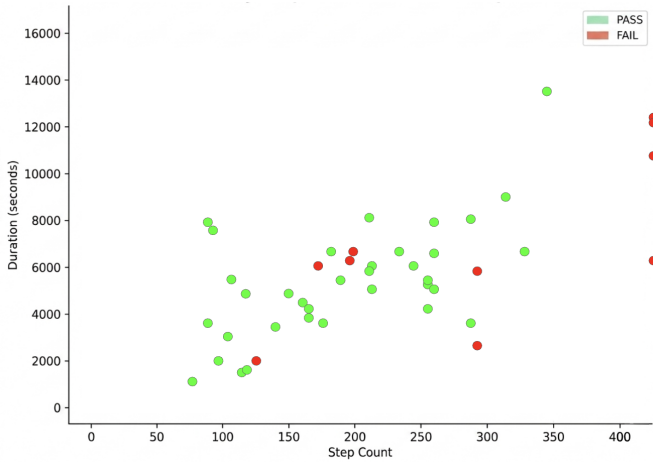
The architecture comparison supports the main design hypothesis. Holding the intermediate representation fixed, GALA outperforms ReAct in both cases. With MiniZinc, GALA reaches 84%, compared with 65% for ReAct, a gain of 19 percentage points. With PySAT / pseudo-Boolean encodings, GALA reaches 93%, compared with 78% for ReAct, a gain of 15 percentage points. This suggests that the GALA-inspired architecture helps because it does not depend on a single modeling trajectory. If one proposal chooses a bad variable schema or misses a constraint, another candidate may still produce a usable reduction.

The intermediate-representation comparison is also clear. PySAT / pseudo-Boolean encodings outperform MiniZinc in every architecture family shown. Under GALA, PySAT / PB reaches 93%, compared with 84% for MiniZinc. Under ReAct, PySAT / PB reaches 78%, compared with 65% for MiniZinc. Under the raw baseline, PySAT / PB reaches 68%, compared with 60% for MiniZinc. This gives PySAT / PB gains of 9, 13, and 8 percentage points respectively. These results support the design choice to use pseudo-Boolean constraints as the main practical IR: they are expressive enough for common SAT constraints, but closer to the solver pipeline than MiniZinc.

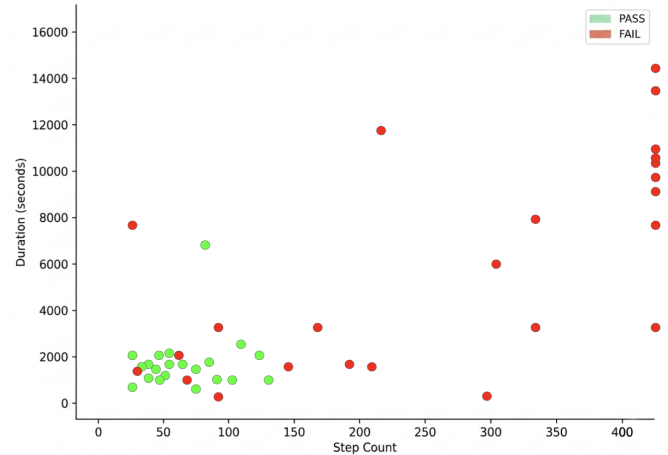
The action-distribution results help explain this difference. In the categorized action chart, MiniZinc-heavy configurations spend a larger share of their actions on implementation and compilation. GALA with MiniZinc uses roughly 70% of its actions on implementation and compilation, while GALA with PySAT / PB uses roughly 30–35% for that category and spends much more effort on planning. ReAct with MiniZinc similarly spends roughly 65–70% on implementation and compilation, while ReAct with PySAT / PB spends closer to 25–30%. This suggests that MiniZinc pushes the agent toward code-writing and debugging, while PySAT / PB lets the agent spend more effort on structured modeling decisions.

The planning, debugging, and implementation/compilation fields are constituted of actual action steps the model can take. What we found is that MiniZinc configurations are dominated by `ADD_MINIZINC_CODE` and model-file update actions, while PySAT / PB configurations rely more on `ADD_MODEL_CONSTRAINTS`, `DEFINE_VARIABLES`, `DEFINE_VARIABLE_PATTERN`, and `ADD_PYTHON_CONSTRAINT_BLOCK`. This matters because the system is not only choosing a solver backend; it is choosing what kind of work the LLM has to perform. MiniZinc can be compact and expressive, but it requires the agent to write and revise more code. PySAT / PB fits the agent pipeline better because the agent can build constraints through structured actions and rely on existing encoders for low-level CNF expansion.

The step-count versus duration plots show the tradeoff between robustness and efficiency. In the ReAct plot, many successful runs cluster between about 30 and 120 steps and below roughly 2,500 seconds. However, as step count grows, failures become more common, especially near the far-right cluster around 400 or more steps, where several runs fail after long durations. This suggests that simply allowing the ReAct agent to take more steps does not reliably fix bad encodings. Long ReAct runs often rep-



(a) GALA: step count vs. duration.



(b) ReAct: step count vs. duration.

Figure 5: Step-count versus duration plots. ReAct has many successful runs clustered at lower step counts, but also several long failed runs. GALA shows successful runs across a wider range of step counts and durations, reflecting broader search over candidate encodings.

resent the agent getting stuck rather than gradually converging.

The GALA plot has a different shape. Successful runs appear across a wider range of step counts and durations, including many successful runs between roughly 100 and 350 steps and several successful runs above 6,000 seconds. This supports the idea that GALA explores a larger encoding space. However, GALA also has expensive failures, including failed runs near 400 or more steps and above 9,000–14,000 seconds. This shows the cost of the architecture: GALA is more robust, but not more efficient. It trades additional LLM calls, candidate generation, and ranking overhead for a better chance of finding a correct encoding.

Overall, the strongest configuration is GALA with PySAT / pseudo-Boolean encodings. GALA improves robustness by exploring multiple candidate reductions before solving, while PySAT / PB makes each candidate easier to integrate, verify, and solve. The main cost is compute. The system spends more time generating and evaluating candidates, but the payoff is a higher pass rate. These results reinforce the main lesson of the project: the bottleneck is usually not the SAT solver, but the quality of the encoding given to it.

10 Surprises and Lessons Learned

One major surprise was that giving the ReAct agent more steps did not necessarily improve encoding quality. More steps seemed like they should allow the agent to repair mistakes, but in practice they often refined the same bad representation. If the initial variable schema or constraint framing was wrong, later edits were usually local patches rather than a true model change.

This shifted the project’s focus from “give the agent more time” to “make sure the agent explores the right modeling space.” A good SAT reduction often depends on choosing the right variables early. This motivated the GALA-inspired design, which

avoids overcommitting to the first model by generating several alternatives.

A second lesson was the distinction between a wrong model and hard search. When a run fails, the solver is not always the problem. The formula may encode a relaxed, incomplete, or over-constrained version of the task. Solver feedback only helps if the formula being solved is actually the intended formula.

The IR experiments also showed that higher-level is not automatically better. MiniZinc is expressive and close to human modeling, but in this implementation it added integration and backend fragility. PySAT and pseudo-Boolean encodings were more practical because they stayed close to SAT while still abstracting away raw CNF.

Finally, simple sanity checks are not enough. A candidate can be syntactically valid and still semantically wrong. Verification must test behavior through micro-instances, SAT/UNSAT oracles, fuzzing, and decoder checks. An LLM-generated model should earn trust by passing tests that connect the formal encoding back to the original problem.

11 Limitations

The system still depends on the LLM producing at least one reasonable encoding. Multi-candidate generation reduces this risk, but if all proposals share the same misunderstanding, the verifier may not recover.

Verification is limited by the tests used. Structural checks, grounding, decoder compatibility, fuzzing, and micro-instances improve confidence, but they are not formal proofs that an encoding matches the natural-language problem. A candidate can pass small tests and still fail on larger or subtler cases.

Some benchmark families remain difficult, especially those requiring global structure, temporal reasoning, or careful state modeling. Path planning, exact cover, Slitherlink, and Sokoban

place heavy demands on variable schema and decoding. Small modeling errors can make the formula too weak, too strong, or difficult to interpret.

MiniZinc support was less reliable than expected. This does not mean MiniZinc is a poor IR generally; it means that, in this implementation, the extra compiler and backend layer introduced fragility. PySAT / pseudo-Boolean encodings were more effective for the current pipeline.

Denabase is promising but should not be overstated as a driver of the final results. Storing verified reductions, fingerprints, gadget patterns, and solver traces is valuable, but in this report Denabase is best framed as a partially integrated memory layer and future direction.

The ranking function is hand-designed. Its signals are reasonable, but the weights may not generalize across problem families. Learning or calibrating these weights from more data would be a natural next step.

Finally, LLM calls dominate runtime. SAT solving itself is usually fast compared with LLM reasoning, so scaling the system is more about reducing or improving LLM calls than optimizing the SAT solver. The evaluation is also limited by benchmark size and diversity, so the results should be interpreted as evidence about this benchmark suite rather than a universal claim about all natural-language-to-SAT modeling.

12 Conclusion and Future Work

This project shows that LLMs can be useful for SAT modeling when embedded in a structured neuro-symbolic pipeline rather than used as direct answer generators. The LLM helps translate natural language into formal encodings, while the SAT solver provides exact symbolic search. In this implementation, the strongest practical approach combined PySAT / pseudo-Boolean encodings, multi-candidate proposal generation, and verifier-gated ranking.

The main lesson is that the bottleneck is often correct modeling, not SAT solving. More ReAct steps did not automatically improve quality, because a bad initial schema often led to refinements of the wrong model. The project also showed that IR choice matters: MiniZinc is attractive, but PySAT / pseudo-Boolean encodings were more practical here. Finally, verification must test semantic behavior, not just syntax.

Future work should make the system more reusable and verifiable. A major next step is to build a memory of correct reductions and encoding patterns. Denabase could be expanded with high-quality verified SAT examples, structural fingerprints, gadget patterns, and solver traces. Then, when the agent sees a new problem, it could retrieve structurally similar encodings and use them as priors instead of solving every problem from scratch.

The verifier and ranking components should also be strengthened. Better micro-instance generation and SAT/UNSAT oracle tests would help distinguish wrong models from hard search. The ranking function could be learned or tuned from data rather than hand-designed. Evaluation should expand to harder and

more diverse benchmarks, especially those requiring global structure, temporal reasoning, and careful decoding.

A final direction is to study the reduction patterns themselves. By analyzing which patterns LLMs reuse and which lead to correct encodings, the project could reveal how language models perform formal modeling. As LLMs improve, this neuro-symbolic approach may become more powerful: the LLM supplies flexible modeling and analogy, while the SAT solver supplies exactness. The long-term goal is not to replace SAT solvers, but to make SAT solving accessible to users who cannot manually write formal encodings.

References

- [1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, 2nd edition. IOS Press, 2021. <https://www.iospress.com/catalog/books/handbook-of-satisfiability-2>
- [2] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006. https://jsatjournal.org/volumes/2/JSAT2_1_Een.pdf
- [3] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python Toolkit for Prototyping with SAT Oracles. SAT, 2018. <https://pysathq.github.io/>
- [4] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. *Principles and Practice of Constraint Programming*, 2007. https://link.springer.com/chapter/10.1007/978-3-540-74970-7_38
- [5] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. ICLR, 2023. https://openreview.net/forum?id=WE_vluYUL-X
- [6] Stefan Szeider. CP-Agent: Agentic Constraint Programming. arXiv:2508.07468, 2025. <https://arxiv.org/abs/2508.07468>
- [7] Junyang Cai, Serdar Kadioglu, and Bistra Dilkina. GALA: Global LLM Agents for Text-to-Model Translation. arXiv:2509.08970, 2025. <https://arxiv.org/abs/2509.08970>
- [8] Kostis Michailidis, Dimos Tsouros, and Tias Guns. CP-Bench: Evaluating Large Language Models for Constraint Modelling. arXiv:2506.06052, 2025. <https://arxiv.org/abs/2506.06052>

- [9] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903, 2022. <https://arxiv.org/abs/2201.11903>
- [10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. 2010. <https://smt-lib.org/papers/smt-lib-reference-v2.0-r10.12.21.pdf>
- [11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Morgan & Claypool, 2012. <https://link.springer.com/book/10.1007/978-3-031-01561-8>