

Beyond Better Helpers: A Shared-Kernel Study of Symbolic Interfaces for Auditable LLM Algebra

Norio Kosaka Genki Osada Shinichi Higashino Shuji Yamaguchi

LY Corporation
Tokyo, Japan

Abstract

Large language models can delegate algebra to external tools, but improvements attributed to a structured tool interface may instead come from the symbolic helper code hidden behind that interface. We isolate these factors on polynomial systems by holding the model, tasks, backend, and symbolic kernel fixed. Using Qwen3-8B, we compare text-only reasoning, unrestricted Python/SymPy generation, generated Python calling the same helper functions as the structured system, and a typed diagnosis–solve–check interface. On 36 tasks, the typed interface obtains 25/36 correct answers, compared with 14/36 for matched-helper code, 17/36 for raw SymPy, and 16/36 for text only. It also produces 23/36 automatically verified-correct answers versus 4/36 for the matched-helper control, and contains no execution-error runs versus 29/36. The largest separation occurs on exact finite-set tasks, where the typed interface solves 8/12 systems and both code conditions solve 0/12. Access to packaged helper functionality alone therefore does not reproduce the typed condition’s performance. The typed interface also removes the need to generate imports, API calls, control flow, and result-handling code, so the comparison measures end-to-end interface usability rather than mathematical reasoning or schema syntax in isolation. We interpret the result as a deployed-interface effect for one model and a development-informed task suite.

1 Introduction

Mathematical language agents increasingly move between natural-language problem statements, formal equations, executable programs, symbolic tools, and machine-checkable conclusions (Yan et al., 2025; Gao et al., 2023; Gou et al., 2024; Pan et al., 2023; Liu et al., 2025). External computation can improve answer accuracy, but execution alone does not make a result auditable. A solver-backed response may still omit solutions, ignore a domain

restriction, hide a failed computation, or report candidates without retaining evidence that supports the final claim (Bagheri Nezhad et al., 2026; Zhang et al., 2024; Liu et al., 2025).

Polynomial systems provide a compact setting in which these obligations can be separated. Candidate assignments can be checked by substitution, structural claims can be assessed with computer algebra, and finite-set answers expose a crucial distinction: zero residuals establish that the listed points are valid, but do not establish that no solutions were omitted (Brake et al., 2016; Sommese et al., 2005; Cox et al., 1997). This makes polynomial systems useful not only for measuring final-answer accuracy, but also for evaluating whether a retained trace validly covers the claim being asserted.

A central experimental difficulty is that tool-interface comparisons often change two things at once. A typed tool may expose constrained arguments and a structured workflow, but also provide normalization, solver wrappers, domain handling, residual computation, and exception handling that a free-form-code agent must synthesize itself. A typed-versus-raw-code comparison therefore cannot determine whether gains come from the model-facing interface or from better packaged symbolic functionality. We address this confound with a *matched-helper code* condition: the model writes ordinary Python, but imports and calls the exact same versioned symbolic kernel used by the typed actions. The primary comparison thus holds helper implementation fixed while changing how the model invokes it and receives results.

We ask three questions. **RQ1** asks whether packaged symbolic helpers improve over unrestricted Python/SymPy generation. **RQ2** asks whether, with the helper implementation fixed, a typed interaction format changes correctness and automatically verified correctness relative to generated Python. **RQ3** asks how the interfaces change ex-

explicit verification, execution failures, runtime, and performance across task families.

Using `qwen3:8b` through Ollama (Qwen Team, 2025; Ollama, 2026), we evaluate four conditions on the same 36 polynomial-system tasks. The typed interface obtains 25/36 correct answers, compared with 14/36 for matched-helper code, 17/36 for raw SymPy, and 16/36 for text only. It yields 23/36 automatically verified-correct answers versus 4/36 for matched-helper code, while reducing runs containing an execution error from 29/36 to 0/36. The largest gap occurs on exact finite-set tasks: the typed interface solves 8/12, whereas both code conditions solve none. Access to the shared helpers alone therefore does not reproduce the typed condition’s performance. The remaining contrast includes constrained invocation, workflow cues, structured statuses, evidence serialization, and the removal of generated glue code; it does not isolate improved mathematical reasoning or any one interface component. Earlier oracle-equation and restricted-action pilots, retained as secondary analyses, further test whether language grounding or the available action surface explains this pattern.

Contributions. This paper makes three contributions:

- **A shared-helper controlled evaluation.** We introduce a matched-helper code baseline that invokes the same symbolic kernel as the typed interface, separating packaged-helper effects from the model-facing interaction format.
- **Claim-aware evaluation.** We distinguish answer correctness, candidate validity, set completeness, explicit verification, execution failure, and automatically verified-correct answers for finite sets, witnesses, and structural diagnoses.
- **Evidence for a deployed-interface effect.** Under a fixed model, task suite, backend, and helper implementation, the typed interface is associated with higher correctness and retained claim support while avoiding the execution failures that dominate generated matched-helper code. This is an end-to-end system result, not evidence that typing alone improves mathematical reasoning.

2 Related Work

Program-aided and neuro-symbolic mathematical reasoning. Program-aided methods separate linguistic decomposition from external computation. PAL delegates computation to generated Python programs (Gao et al., 2023), Program of Thoughts uses executable programs to separate reasoning from calculation (Chen et al., 2023), and ToRA interleaves language-model reasoning with external mathematical tools (Gou et al., 2024). Related neuro-symbolic systems translate natural language into formal representations executed by a solver, including Logic-LM (Pan et al., 2023) and LINC (Olausson et al., 2023). MCP-Solver similarly exposes constraint, SAT, and SMT systems through a standardized tool interface (Szeider, 2025). These works establish the value of external symbolic computation. Our focus is complementary: rather than comparing tool use with no tool use alone, we ask whether the model-facing interface matters when the symbolic helper implementation is held fixed.

Verifiable code and symbolic checking. SymCode prompts an LLM to generate self-contained SymPy programs, executes them, and uses interpreter feedback for verification and self-debugging (Bagheri Nezhad et al., 2026). Our raw-code condition follows this general reasoning-as-code paradigm, while our matched-helper condition provides a stronger control: generated Python can call the same normalization, diagnosis, solving, domain, and checking functions used by the typed interface. Work on symbolic verification of intermediate reasoning likewise demonstrates the value of machine-checkable representations for process evaluation (Zhang et al., 2024). SAFE uses Lean to verify step-level mathematical claims (Liu et al., 2025), and tool-integrated Lean validation has also been used to generate and align synthetic mathematical data (Obozov et al., 2025). Our setting is less expressive than a proof assistant, but permits exact candidate checks and explicit completeness obligations for a controlled class of algebraic claims.

Tool-using agents and structured action spaces. ReAct formalized the reasoning–action–observation loop (Yao et al., 2023), while Toolformer and Gorilla studied tool invocation and API use (Schick et al., 2023; Patil et al., 2023). Structured actions can reduce syntax and serialization failures, but comparisons against

raw code usually confound action format with the functionality packaged behind the API. Our four-condition design decomposes this comparison. Raw SymPy versus matched-helper code probes the effect of packaged symbolic functionality; matched-helper code versus typed actions probes the deployed interaction format with the helper kernel fixed. The latter still jointly changes schema constraints, workflow guidance, status presentation, and evidence serialization, so we do not interpret it as a schema-syntax-only intervention.

Positioning within MathNLP. The tasks require a model to interpret variables, equations, domains, and answer modes before producing a checkable mathematical claim. The paper is therefore most closely related to neuro-symbolic mathematical inference and evaluation of tool-supported mathematical reasoning. It does not propose a new computer-algebra algorithm or a full autoformalization system. Its contribution is a controlled evaluation of the interface layer through which mathematical language agents exchange formal problems, candidate solutions, structural statuses, and verification evidence with symbolic computation.

3 Task Setting and Symbolic Interfaces

Problem setting. A task consists of a mathematical prompt u , a variable list $V = (x_1, \dots, x_n)$, a declared solution domain $D \in \{\mathbb{R}, \mathbb{C}\}$, and polynomial equations

$$F = \{f_i(x_1, \dots, x_n) = 0\}_{i=1}^m. \quad (1)$$

The required answer mode is one of the following:

- **ALL-SOLUTIONS:** return the complete finite solution set in D ;
- **WITNESS:** return at least one assignment satisfying every equation and domain restriction; or
- **DIAGNOSIS:** return a domain-aware structural label.

For diagnosis tasks, algebraic inconsistency is distinguished from absence of real solutions: a system may have complex solutions but no point in $\mathbb{R}$. The present study isolates tool-mediated symbolic reasoning after the task variables, equations, domain, and answer mode have been specified; it does not evaluate open-ended autoformalization.

Verification obligations. For a returned assignment s , *candidate validity* requires $f_i(s) = 0$ for every original equation and satisfaction of the declared domain. This is sufficient for an **WITNESS** claim. For an **ALL-SOLUTIONS** answer $\hat{S}$, candidate validity establishes only soundness of the listed points. The claim additionally requires *set completeness*: every solution in D must belong to $\hat{S}$. Consequently, residual checks do not by themselves establish an all-solutions claim. A run is counted as verified-correct only when its final answer is correct and its retained evidence covers the obligations associated with its answer mode.

Shared symbolic kernel. The matched-helper and typed conditions invoke one versioned symbolic kernel. The kernel implements equation normalization, structural diagnosis, candidate generation, domain filtering, residual computation, and candidate checking. It separates execution, candidate-validity, domain-validity, diagnosis, and completeness statuses. In particular, successful execution or zero residuals for returned candidates are not automatically treated as evidence that a finite solution set is complete.

Model-facing interfaces. The raw-code condition writes unrestricted Python/SymPy and is responsible for constructing solver calls, checks, and output handling. The matched-helper condition also writes ordinary Python, but may import the shared kernel functions directly. The typed condition accesses the same kernel through three schema-constrained actions:

1. `diagnose_groebner`, which returns supported structural conclusions such as inconsistency or zero-dimensionality;
2. `solve_polynomial_system`, which returns normalized candidates, domain annotations, residual information, and a conservative completeness status; and
3. `check_solution`, which substitutes a proposed assignment into the original equations and reports symbolic residuals and domain validity, but never certifies completeness of an enumerated set.

The typed adapter performs request validation and structured serialization, but delegates the mathematical operations to the same kernel used by matched-helper code.

Trace semantics. Every tool-enabled run retains generated code or typed requests, helper inputs and outputs, structured statuses, warnings, errors, and the final response. Execution failures and unsupported cases remain visible rather than being converted into ordinary answers. Evidence is linked to the final claim rather than credited merely because a solver or checker was invoked.

4 Experimental Design

Research questions. We study three questions. **RQ1** asks whether providing a shared symbolic helper layer improves over unrestricted Python/SymPy generation. **RQ2** asks whether, with helper functionality fixed, typed invocation changes answer correctness and automatically verified correctness relative to generated Python. **RQ3** asks how interface design changes explicit verification, execution failures, runtime, and performance across task families.

Conditions. We evaluate four conditions on every task:

1. **Text only** (`text_only`): the model receives the task and no external computation.
2. **Raw SymPy** (`raw_sympy`): the model writes and executes unrestricted Python/SymPy.
3. **Matched-helper code** (`matched_helper_code`): the model writes ordinary Python that imports the same normalization, diagnosis, solving, domain, and checking functions used by the typed condition.
4. **Typed helper interface** (`typed_helper_interface`): the model invokes schema-constrained diagnosis, solve, and check actions backed by the shared kernel and receives structured status and evidence records.

All conditions receive the same equations, variables, domain, answer-mode specification, and final-answer requirement. Conditions 2–4 use SymPy as the underlying algebra system. Conditions 3 and 4 additionally use the exact same versioned symbolic-kernel implementation. Their intended difference is the model-facing interaction format: generated Python and free-form result handling versus validated action calls and structured responses. Workflow wording, status presentation, and evidence serialization necessarily differ with

this format and are therefore part of the deployed-interface treatment. The study does not include an additional code condition supplied with a working import and invocation template; consequently, it does not separate low-level API scaffolding from the broader structured interaction protocol.

Primary contrasts. Raw SymPy versus matched-helper code probes whether packaged symbolic functionality is sufficient to improve generated-code reasoning. Matched-helper code versus the typed interface is the primary controlled contrast: helper implementation is fixed, while invocation, workflow affordances, and result serialization change. Text only provides context for the value of external computation rather than a matched causal control.

Model and execution environment. All runs use `qwen3:8b` through Ollama (Qwen Team, 2025; Ollama, 2026), with temperature 0, seed 0, an 8192-token context, a 4096-token output limit, at most eight calls per task, and a 60-second timeout. The canonical run record stores the exact model digest, chat-template hash, prompt and schema hashes, symbolic-kernel version, dependency versions, hardware metadata, and raw model/tool traces.

Benchmark. The benchmark contains 36 tasks: 12 exact zero-dimensional systems, 8 hidden-resolution or witness tasks, 8 structural-diagnosis tasks, and 8 geometry-inspired algebraic systems. The systems contain at most four variables and have maximum degree at most five. Each task record includes its prompt, formal equations, variable order, declared domain, answer mode, task family, and gold answer. Gold records were checked with SymPy and a separately recorded derivation. The suite was frozen before the fresh four-condition rerun; because it also informed earlier pilot development, we treat it as a controlled evaluation suite rather than a pristine held-out benchmark.

Prompts and budgets. No condition uses few-shot examples or task-specific prompt tuning. The task content and final-answer instructions are shared. Condition-specific text only describes the available interaction mechanism: unrestricted Python, the matched helper API, or typed action names and payloads. Tool-enabled conditions use the same maximum-call and timeout policy.

Correctness. Scoring is answer-mode specific. Exact finite tasks require equality with the complete gold solution set up to variable order and exact algebraic equivalence in the declared domain. Witness tasks require at least one valid assignment. Diagnosis tasks require the correct domain-aware structural label. Exact symbolic comparison is preferred; numerical fallback is used only when exact simplification is inconclusive, with residual tolerance 10^{-6} . The scorer version and task-level outputs are frozen in the canonical result record.

Verified correctness. We separately report explicit verification and *verified-correct* answers. Explicit verification records whether the model executed a post-candidate check. Verified correctness is stricter: the final answer must be correct and the retained trace must contain valid evidence covering the claim. For a witness, a claim-linked substitution and domain check suffice. For an all-solutions answer, candidate checks must be accompanied by supported completeness evidence. These labels are produced by the frozen automatic scorer under a condition-independent rubric; because structured traces are easier to parse, we discuss scorer endogeneity separately.

Operational outcomes. A run is marked as containing an *execution error* when its retained trace records a generated-code, helper-invocation, backend, timeout, or tool-execution failure. This is a process metric: a run may contain an error and still recover a correct final answer. The aggregate includes failures in syntax or imports, helper arguments, generated control flow, backend execution, timeouts, and result handling; it must not be read as a count of mathematical solver errors. We also report wall-clock runtime. Failed calls, malformed answers, and unsupported cases remain in the denominator.

Paired analysis. Every condition is evaluated on the same tasks, so the primary comparisons are paired. We report task-level counts and absolute paired differences. The canonical analysis additionally records the 2×2 paired contingency tables, exact McNemar tests, and task-paired bootstrap confidence intervals; subset results are treated as descriptive because of their small sizes.

Reproducibility. All tables are generated from canonical task-level records rather than hand-written aggregates. The companion artifact contains the task and gold manifests, exact prompts,

action schemas, shared-kernel implementation, model and environment metadata, raw traces, scoring code, and table-generation commands.

Secondary legacy analyses. To contextualize the fresh controlled comparison, we retain four earlier pilot analyses: a diagnostic-only subset, an oracle-equation presentation that removes language-to-equation ambiguity, and solve-only and check-only restrictions of the typed action surface. These batches predate the canonical shared-helper rerun and have partially documented orchestration differences. We therefore use them only as exploratory mechanism analyses, do not pool them with the fresh batch, and report their full aggregates in Appendix B. Their legacy “evidence” label denotes evidence attachment rather than the stricter verified-correct metric used in the main experiment.

5 Results

Primary shared-helper comparison. With the symbolic helper implementation fixed, the typed interface obtains 25/36 correct answers, compared with 14/36 for matched-helper code, an absolute difference of 11 tasks or 30.6 percentage points. Automatically verified-correct answers increase from 4/36 to 23/36, a difference of 19 tasks or 52.8 points. Thus access to packaged normalization, solving, domain, and checking functions alone does not reproduce the typed condition’s performance. The remaining treatment is the deployed interaction format, including constrained invocation, workflow cues, structured statuses, evidence serialization, and the removal of generated glue code; the experiment does not isolate any one of these components or a change in mathematical reasoning.

Packaged helpers are not sufficient. Raw SymPy reaches 17/36 correct answers, whereas matched-helper code reaches 14/36. Simply making the shared kernel importable from generated Python does not improve this model’s performance. Instead, matched-helper code contains an execution error in 29/36 runs, compared with 7/36 for raw SymPy and none for the typed interface. This pattern indicates that a capable symbolic API is not sufficient for reliable end-to-end use when the model must synthesize valid invocation code, control flow, and result handling. At the same time, the high matched-helper error rate makes glue-code burden a plausible major source of the correctness

Method	Correct	Verified-correct	Explicit verification	Execution-error runs	Avg. time (s)
Text only	16/36 (44.4%)	0/36 (0.0%)	0/36 (0.0%)	0/36 (0.0%)	7.03
Raw SymPy	17/36 (47.2%)	5/36 (13.9%)	14/36 (38.9%)	7/36 (19.4%)	12.63
Matched-helper code	14/36 (38.9%)	4/36 (11.1%)	4/36 (11.1%)	29/36 (80.6%)	42.98
Typed helper interface	25/36 (69.4%)	23/36 (63.9%)	23/36 (63.9%)	0/36 (0.0%)	18.69

Table 1: Fresh four-condition results on the 36-task suite. Verified-correct requires a correct final answer and valid evidence covering the answer-mode claim. Explicit verification records a separate post-candidate check. An execution-error run contains at least one recorded code, helper, backend, timeout, or tool-execution failure; it may still end with a correct answer.

Method	Exact	Hidden	Diagn.	Geom.
Raw SymPy	0/12	7/8	8/8	2/8
Matched-helper	0/12	4/8	8/8	2/8
Typed interface	8/12	6/8	8/8	3/8

Table 2: Correct answers by subset for the three tool-enabled conditions. The small subset sizes support descriptive, not broad population, conclusions.

gap. We therefore treat the result as an interface-usability comparison rather than evidence of improved mathematical reasoning, and we do not re-label every failure as a mathematical solver error.

Correctness and retained evidence. The typed condition produces 23 verified-correct answers, close to its 25 correct answers, whereas raw SymPy and matched-helper code produce only 5 and 4, respectively. The equality between explicit-verification and verified-correct counts in the typed aggregate is empirical, not definitional: a check receives verified-correct credit only when the final answer is correct and the evidence covers the relevant claim. In particular, substitution checks alone do not establish completeness of an all-solutions answer. These numbers therefore measure automatically auditable claim support under the frozen scorer, not the truthfulness of an unrestricted chain of thought.

Where the difference occurs. The largest separation occurs on exact zero-dimensional systems. The typed interface solves 8/12, while both code conditions solve 0/12. These tasks combine enumeration, domain handling, and complete final-answer construction; coordinating those obligations is one plausible source of the observed gap, but the current design does not identify the mechanism. All three tool-enabled conditions solve the eight diagnosis tasks, making that subset a sanity check rather than evidence of an interface advantage. Raw SymPy is strongest on hidden-

solution tasks (7/8 versus 6/8 typed), where finding one valid witness can reward flexible exploration. Geometry-inspired systems remain difficult for every condition (3/8 typed versus 2/8 for each code condition).

Runtime and failure behavior. Matched-helper code is both the least accurate tool-enabled condition and the slowest, averaging 42.98 seconds, consistent with repeated generated-code or helper-invocation failures. The typed interface averages 18.69 seconds: slower than raw SymPy, but substantially more accurate and more often verified-correct. Runtime should therefore be interpreted jointly with failure and answer quality rather than as a standalone efficiency ranking.

Secondary mechanism analyses. The legacy pilots provide complementary, non-confirmatory evidence (Appendix B). Supplying equations, variables, and domains explicitly preserved the earlier raw-versus-typed separation (19/36 versus 27/36), suggesting that equation extraction was not its sole source. The diagnostic-only subset was saturated at 8/8 for text, raw code, and typed tools. Within a separate action-surface batch, solve-only reached 22/36 correct, full typed 27/36, and check-only 28/36; check-only also produced the highest legacy evidence-attachment count (31/36), but more error-containing runs than full typed (6/36 versus 1/36). These point estimates motivated the verification-first hypothesis, but different retained batches and the weaker legacy evidence metric prevent pooling or causal interpretation.

Interpretation. The fresh experiment supports a narrow but stronger conclusion than the legacy three-condition comparison. Under one model, one development-informed 36-task suite, one backend, and an identical symbolic kernel, the model-facing interface is associated with substantial changes in correctness, automatically auditable evidence, and execution reliability. The result is not a claim that

type annotations improve mathematical reasoning, that any single interface component causes the difference, or that the typed workflow generalizes beyond this model–benchmark combination.

6 Discussion

The gain persists after matching helper functionality. The matched-helper control resolves the main confound in the original typed-versus-raw comparison. Both matched-helper code and the typed interface invoke the same versioned symbolic kernel, yet the typed condition is 30.6 percentage points more accurate and 52.8 points higher in automatically verified correctness. Access to the same normalization, solving, checking, and domain-handling code therefore does not reproduce the typed condition’s performance. This supports a deployed-interface effect under the present model and tasks, but not a claim that the typed condition improves the model’s underlying mathematical reasoning.

The treatment is broader than schema syntax. The matched comparison does not identify a single causal primitive. Typed invocation changes argument constraints, workflow guidance, status presentation, error exposure, and evidence serialization together. The paper’s claim is consequently about the deployed mathematical interaction format, not about JSON schemas or static types in isolation. Separating these components would require additional factorial interventions, such as neutralizing workflow language while retaining typed calls or giving generated code an equivalent structured response object.

An executable-template control remains open. The matched-helper prompt documents the API, but it does not provide a minimal working program that demonstrates the correct imports and helper invocations. Such a control could reduce low-level code-generation failures while retaining ordinary Python control flow and would help distinguish API scaffolding from the broader structured protocol. Because we did not run this condition, the present study cannot determine whether the typed advantage would remain after that burden is removed. The high matched-helper error rate makes this an especially important target for replication.

A helper API is not automatically usable by an LLM. The matched-helper condition performs worse than raw SymPy and records errors in 29/36

runs. This is a substantive finding: exposing a reliable mathematical library does not ensure that a model can generate correct imports, arguments, control flow, and final-answer serialization. The typed interface removes much of this glue-code burden, which is part of the deployed system rather than a nuisance external to it. However, the magnitude of the error rate also makes baseline usability a central threat to mechanistic interpretation. The aggregate error indicator spans syntax and imports, invalid calls, generated control flow, backend failures, timeouts, and output handling; it should not be read as a count of mathematical solver failures.

Correctness and evidence should be evaluated jointly. A correct answer may be unsupported, while a valid check may cover only one part of an incorrect or incomplete claim. This distinction is especially important for finite solution sets: verifying every returned point establishes candidate validity but not set completeness. The large typed-interface gain in verified correctness suggests that structured traces can make mathematical claims easier to audit, but this metric is partly endogenous because the typed responses are designed for machine parsing. Condition-blind human validation would provide a stronger test of evidence relevance and coverage.

Task structure determines which interface helps. The largest observed separation occurs on exact finite-set tasks, where the model must coordinate enumeration, domain restrictions, completeness status, and final serialization. This pattern suggests, but does not establish, that coordination and serialization burden contribute to the gap. Raw SymPy remains competitive on witness tasks, where one valid point is enough and free-form exploration can be useful. The diagnostic subset is saturated, and the geometry-inspired subset remains difficult. The result is therefore not that structure uniformly improves every mathematical task, but that its value depends on the obligations of the requested claim.

Legacy pilots sharpen, but do not establish, the mechanism. The earlier oracle-equation analysis preserved the typed advantage after the formal system was supplied directly, making language-to-equation extraction an insufficient explanation for that batch. The action-surface pilot found that a check-only interface matched or slightly exceeded the full typed workflow in correctness and legacy evidence attachment, whereas the full workflow

produced fewer error-containing runs. Together these results motivated the present verification-first framing: explicit checking may improve claim support, while a broader action surface may improve operational robustness. Because these pilots used separately retained batches and the old evidence-attachment metric, they are secondary hypothesis-generating analyses rather than confirmation of a specific causal component.

Implications for MathNLP. The study isolates an often hidden layer between mathematical language understanding and symbolic computation: the protocol through which variables, equations, domains, candidates, statuses, and evidence are exchanged. MathNLP evaluations should treat this protocol as an experimental variable rather than attributing all tool-use gains to either the language model or the solver. A broader benchmark should additionally vary linguistic realization—direct equations versus prose, aliases, reordered constraints, implicit domains, coreference, and distractors—to separate language grounding from symbolic interaction.

7 Conclusion

We studied whether the model-facing interface to symbolic algebra matters after controlling for the helper implementation behind it. On 36 polynomial-system tasks, a typed diagnosis–solve–check interface obtains 25/36 correct and 23/36 automatically verified-correct answers. Generated Python calling the same symbolic kernel obtains 14/36 and 4/36, respectively, and contains an execution error in 29/36 runs, compared with none for the typed condition. Raw SymPy and text-only reasoning reach 17/36 and 16/36 correct answers.

Access to the same symbolic functions does not reproduce the typed condition’s performance. The typed condition also removes the burden of generating imports, API calls, control flow, and result-handling code, while changing workflow guidance, error reporting, and evidence serialization. The defensible result is therefore an end-to-end deployed-interface effect under one model and a small, development-informed task suite—not evidence that schema syntax improves mathematical reasoning. Whether the advantage remains when generated code is given a minimal executable template is left unresolved.

For mathematical NLP, symbolic functionality and its model-facing protocol should be evaluated

separately, and final-answer accuracy should be reported together with claim-aware verification and observable failure behavior.

8 Limitations

Bundled interface effect. The matched-helper control holds the symbolic kernel fixed, but the typed condition still jointly changes schema constraints, workflow guidance, status presentation, evidence serialization, and the need to generate glue code. The results therefore support an end-to-end effect of the deployed interface, not an effect of schema syntax or improved mathematical reasoning in isolation.

No executable-template control. We did not give matched-helper code a minimal working template demonstrating correct imports and helper invocations. Such scaffolding could reduce its high execution-error rate and would provide a cleaner test of the broader structured protocol after low-level API usage is matched. The present comparison cannot predict whether the observed advantage would persist under that control.

Evaluation scope. The study uses one 8B model, one symbolic backend, and 36 small polynomial-system tasks, with subsets of only 8–12 tasks. The benchmark was fixed before the final four-condition evaluation, but had informed earlier development. Subset comparisons are therefore descriptive, and replication with additional models, symbolic systems, independently held-out tasks, and more linguistically diverse problems is needed to establish broader generality.

Errors and verification. Matched-helper code produces execution errors in 29/36 runs. This aggregate combines code-generation, invocation, backend, timeout, and output-handling failures; it is not a mathematical-error taxonomy, and its magnitude limits mechanistic interpretation of the correctness gap. Moreover, candidate substitution establishes validity but not completeness of an ALL-SOLUTIONS claim. Verified correctness follows an automatic, interface-sensitive scorer: structured typed outputs are easier to parse, and no condition-blind human audit was conducted. Unsupported cases receive no completeness credit.

References

- Sina Bagheri Nezhad, Yao Li, and Ameeta Agrawal. 2026. [SymCode: A neurosymbolic approach to mathematical reasoning via verifiable code generation](#). In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 1489–1503, Rabat, Morocco. Association for Computational Linguistics.
- Daniel A Brake, Jonathan D Hauenstein, and Alan C Liddell Jr. 2016. Validating the completeness of the real solution set of a system of polynomial equations. In *Proceedings of the 2016 ACM International Symposium on Symbolic and Algebraic Computation*, pages 143–150.
- Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.
- David Cox, John Little, Donal O’shea, and Moss Sweedler. 1997. *Ideals, varieties, and algorithms*. Springer.
- Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.
- Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujia Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2024. ToRA: A tool-integrated reasoning agent for mathematical problem solving. In *International Conference on Learning Representations*.
- Chengwu Liu, Ye Yuan, Yichun Yin, Yan Xu, Xin Xu, Zaoyu Chen, Yasheng Wang, Lifeng Shang, Qun Liu, and Ming Zhang. 2025. [SAFE: Enhancing mathematical reasoning in large language models via retrospective step-aware formal verification](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12171–12186, Vienna, Austria. Association for Computational Linguistics.
- Mark Obozov, Michael Diskin, Aleksandr Beznosikov, Alexander Gasnikov, and Serguei Barannikov. 2025. [Synthetic proofs with tool-integrated reasoning: Contrastive alignment for LLM mathematics with lean](#). In *Proceedings of the 3rd Workshop on Mathematical Natural Language Processing*, pages 195–202, Suzhou, China. Association for Computational Linguistics.
- Theo Olausson, Alex Gu, Benjamin Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua B. Tenenbaum, and Roger Levy. 2023. [LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5153–5176, Singapore. Association for Computational Linguistics.
- Ollama. 2026. Ollama. <https://github.com/ollama/ollama>. Version 0.32.0, accessed July 2026.
- Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. 2023. [Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning](#). In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore. Association for Computational Linguistics.
- Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large language model connected with massive APIs. *arXiv preprint arXiv:2305.15334*.
- Qwen Team. 2025. [Qwen3 technical report](#). *arXiv preprint arXiv:2505.09388*.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*.
- Andrew J Sommese, Charles W Wampler, et al. 2005. *The Numerical solution of systems of polynomials arising in engineering and science*. World Scientific.
- Stefan Szeider. 2025. [MCP-Solver: Integrating language models with constraint programming systems](#). *arXiv preprint arXiv:2501.00539*.
- Yibo Yan, Jiamin Su, Jianxiang He, Fangteng Fu, Xu Zheng, Yuanhuiyi Lyu, Kun Wang, Shen Wang, Qingsong Wen, and Xuming Hu. 2025. A survey of mathematical reasoning in the era of multimodal large language model: Benchmark, method & challenges. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 11798–11827.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*.
- YiFan Zhang, Hanlin Zhang, Li Li, and Eric Xing. 2024. [Evaluating step-by-step reasoning through symbolic verification](#). In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 2984–3002, Mexico City, Mexico. Association for Computational Linguistics.

A Fresh Shared-Helper Experiment

The confirmatory experiment evaluates the four conditions in Section 4 on the same 36 tasks. The main analysis uses only this fresh batch. Earlier three-condition, oracle-equation, solve-only, and check-only runs motivated the controlled design but are not pooled with the fresh results or used for the paper’s primary claims.

A.1 Shared-kernel equivalence

The matched-helper and typed conditions reference the same symbolic-kernel version and helper implementations. The typed actions are thin adapters that validate requests, call the kernel, and serialize its response. The artifact records the helper or kernel hash for each run, generated Python for code conditions, typed requests and responses, normalized equations, variable order, coefficient field, solution domain, and returned execution, candidate, domain, diagnosis, and completeness statuses.

A.2 Exact prompts and budgets

The artifact includes the verbatim system and user messages for all four conditions, the helper API documentation shown to matched-helper code, the exact action schemas shown to the typed condition, and all resolved generation and execution settings. The fresh batch uses temperature 0, seed 0, an 8192-token context, a 4096-token output limit, at most eight calls per task, and a 60-second timeout. Prompt, schema, task, scorer, and kernel hashes are stored in the run manifest.

A.3 Benchmark and gold records

Each task record includes:

- task identifier and family;
- prompt and formal equations;
- variable order, coefficient assumptions, and requested domain;
- answer mode;
- number of variables, maximum degree, and coefficient type;
- gold answer and recorded validation path; and
- construction source and development-exposure metadata.

The 36-task suite was frozen before the fresh rerun, but its prior use in pilot development is explicitly recorded.

A.4 Scoring and evidence obligations

The scorer documents final-answer extraction, exact algebraic equivalence, numerical fallback precision and tolerance, domain validation, diagnosis-label mapping, and the distinction between candidate validity and finite-set completeness. For each task-condition pair, it records final correctness, explicit verification, evidence presence, relevance, validity, coverage, verified correctness, execution status, error category, runtime, and trace identifier. A check invocation alone does not imply verified correctness.

A.5 Paired task-level analysis

All conditions are evaluated on the same tasks. The released task-level table therefore supports the primary matched-helper-versus-typed comparison through its 2×2 paired contingency table, discordant counts, exact McNemar test, absolute paired difference, and task-paired bootstrap interval. The same analysis is provided for raw SymPy versus matched-helper code and raw SymPy versus the typed interface. Subset analyses are reported descriptively.

A.6 Execution-failure taxonomy

The aggregate execution-error indicator is intentionally broad and does not mean that every failure originates in SymPy. The task-level artifact separates at least generated-code syntax, prohibited or failed import, incorrect helper signature or argument, helper exception, backend exception, timeout, unsupported algebra, output-parsing failure, missing final answer, domain mistake, incomplete solution set, and evidence-answer mismatch. Runs with an error remain in the denominator and may still receive answer-correctness credit if they recover a correct final response.

B Legacy Pilot Analyses

The following results recover the exploratory batches that preceded the fresh shared-helper experiment. They are included for transparency and mechanism analysis, but are not pooled with the fresh results. In all tables below, *Evidence* is the original evidence-attachment label—a trace contained residuals, substitution, or structured solver output linked by the legacy scorer—not the stricter verified-correct outcome used in the main paper. Likewise, *Err:* indicates a run containing a tool

or execution error and may overlap with a correct final answer.

B.1 Legacy three-condition comparison

The initial study compared text-only reasoning, unrestricted SymPy code, and the original full typed workflow. It established the descriptive interface gap that motivated the matched-helper control, but bundled helper implementation with invocation format.

B.2 Language-grounding and diagnosis analyses

The oracle presentation supplied equations, variables, and domains explicitly to the tool-enabled conditions. The typed advantage remained descriptive (27/36 versus 19/36), indicating that equation extraction alone did not explain the legacy gap. The diagnostic-only subset was saturated for all conditions and therefore functioned as a sanity check rather than a discriminative benchmark.

B.3 Action-surface analysis

A separately retained batch restricted the typed action surface. Solve-only removed explicit candidate checking, while check-only required the model to propose assignments before verification. Check-only achieved the largest legacy correctness and evidence-attachment point estimates, whereas the full workflow had the fewest error-containing runs. These results motivated the verification-first hypothesis, but the full-workflow row differs from the legacy main batch in explicit-check rate and mean calls; the rows are therefore not interchangeable.

B.4 How the pilots relate to the main experiment

The fresh four-condition study supersedes the legacy main comparison for the paper’s central claim because it adds the shared-kernel matched-helper control and uses canonical provenance and stricter verified-correct scoring. The legacy analyses retain a narrower role: the oracle batch tests whether formal-system extraction alone explains the earlier gap, the diagnostic batch identifies a saturated subset, and the action-surface batch motivates explicit verification as a candidate mechanism. None isolates schema constraints, workflow wording, status presentation, or evidence serialization individually.

C Reproducibility Record

The companion artifact contains:

- the exact model identifier, digest, chat template, and reasoning mode;
- Ollama, Python, SymPy, operating-system, and hardware versions;
- decoding, context, token, call, retry, and timeout settings;
- canonical task, gold, run-config, tool-call, run, and score records;
- raw model messages, generated code, typed requests, and tool traces;
- the frozen scorer and table-generation code; and
- environment files and commands that regenerate every reported table.

Method	Correct	Evidence	Explicit check	Error runs	Audit abst.	Calls	Time (s)
Text only	15/36	0/36	0/36	0/36	21/36	0.00	8.76
Raw SymPy	17/36	7/36	17/36	9/36	18/36	1.69	12.53
Full typed workflow	27/36	27/36	10/36	1/36	9/36	2.03	10.98

Table 3: Legacy main-batch aggregates. This comparison predates the matched-helper control and therefore does not isolate the deployed interaction format from packaged helper functionality.

Setting	Method	Correct	Evidence	Explicit	Err.	Abst.	Calls	Time (s)
Diagnostic (8)	Text only	8/8	0/8	0/8	0/8	0/8	0.00	8.02
Diagnostic (8)	Raw SymPy	8/8	0/8	0/8	2/8	0/8	1.25	7.87
Diagnostic (8)	Typed interface	8/8	1/8	0/8	1/8	0/8	1.12	6.44
Oracle (36)	Raw SymPy	19/36	9/36	20/36	12/36	17/36	1.81	15.41
Oracle (36)	Typed interface	27/36	27/36	12/36	5/36	8/36	2.25	19.44

Table 4: Legacy diagnostic-only and oracle-equation analyses. The oracle batch reduces language-grounding ambiguity but still bundles helper functionality and interface format.

Action surface	Correct	Evidence	Explicit check	Err.	Abst.	Calls	Time (s)
Full typed workflow	27/36	27/36	6/36	1/36	9/36	1.86	10.96
Solve-only	22/36	27/36	0/36	5/36	14/36	1.00	7.38
Check-only	28/36	31/36	36/36	6/36	8/36	1.75	8.95

Table 5: Legacy action-surface pilot. Point estimates are descriptive and use the legacy evidence-attachment metric. They are not pooled with the fresh shared-helper experiment.