PrologMCP: A Standardized Prolog Tool Interface for LLM Agents

Agnieszka Mensfelt[1], Adarsh Prabhakaran[1], Adrian Haret[1], Vince Trencsenyi[1] and Kostas Stathis[1]

[1]Department of Computer Science, Royal Holloway, University of London, UK

Abstract

Frontier reasoning-tuned language models still fail on deductive tasks at depth, and the cost of improved performance through extended internal reasoning scales poorly. Symbolic delegation offers a complementary route: a language model translates the problem, while a solver performs the inference. However, current autoformalization pipelines for logic programming are typically bespoke integrations tied to particular tasks or agents. We introduce PrologMCP, a task-agnostic, open-source server that exposes Prolog as a stateful tool through the Model Context Protocol (MCP). Its compact tool interface, structured error reporting, and per-session isolation make the *translate–run–inspect–repair* loop a reusable primitive for MCP-capable agents. We evaluate a formalizer agent enhanced with PrologMCP against standard and reasoning LLMs (Claude Sonnet 4.6, GPT-4.1, and o4-mini) on two subsets of PARARULE-Plus: a general-purpose sample and a more challenging one targeting a specific failure mode of natural-language reasoning. On the general sample, the formalizer matches or exceeds reasoning LLMs (accuracy 1.00 vs. 1.00 / 0.998), with the largest gains over standard models (0.762 for GPT-4.1). On the challenging subset, the formalizer remains near-perfect (1.00 / 0.99) while reasoning LLMs drop to 0.95 / 0.94. These results suggest that delegating inference to Prolog via MCP is a robust and inspectable alternative to extended natural-language reasoning.

Keywords

Prolog, Model Context Protocol, Autoformalization, Large Language Models, Symbolic Reasoning

1. Introduction

Since the dawn of artificial intelligence, reasoning has been considered a constitutive component of intelligent behaviour [1, 2, 3]. Large language models, after initially underperforming on reasoning benchmarks, have made rapid progress; however, important limitations remain.

First, accuracy degrades on long deductive chains, even for strong reasoning-tuned models. Unlike a symbolic solver, for which multi-step rule application preserves correctness by construction, a language model has no mechanism ensuring that each generated step is a valid logical consequence of the previous one. This lack of stepwise guarantees makes long chains fragile: errors can accumulate, and successful reasoning depends on maintaining a coherent derivation. Lin et al. [4] demonstrate this collapse on grid-style logic puzzles, where frontier reasoning models suffer significant accuracy drops as problem complexity increases. Kazemi et al. [5] likewise report that frontier reasoning models remain far from saturation on a hard reasoning benchmark. Earlier analyses of compositional tasks [6] and template-based variants of standard mathematics benchmarks [7] attribute such failures to pattern matching over training-distribution structure rather than genuine multi-step deduction, with accuracy degrading as the number of required reasoning steps grows.

Second, the cost of additional reliability scales poorly. Current reasoning-tuned models attempt to improve reliability by allocating more internal reasoning, or *thinking*, tokens. However, this increases inference cost while providing no guarantee that the additional tokens correspond to valid deductive steps. On tasks whose rules are simple but whose search space is combinatorially branching, additional generation can therefore yield diminishing returns. Recent work characterising these limits [8, 4]

Joint Workshop on Statistics and Knowledge Integration for Logic, Learning, Ethical Decisions, and LLMs (SKILLED-LLMs 2026), co-located with FLoC 2026, Lisbon, Portugal

agnieszka.mensfelt@rhul.ac.uk (A. Mensfelt); adarsh.prabhakaran@rhul.ac.uk (A. Prabhakaran); adrian.haret@rhul.ac.uk (A. Haret); vince.trencsenyi@rhul.ac.uk (V. Trencsenyi); kostas.stathis@rhul.ac.uk (K. Stathis)

finds that increasing the thinking budget yields diminishing returns beyond a problem-complexity threshold, with reasoning models in some cases even reducing their thinking-token usage as problems become harder. By contrast, symbolic solvers—Prolog for rule-based deduction, SAT and CSP solvers for combinatorial search—routinely handle problems many orders of magnitude beyond the thresholds at which reasoning-tuned models collapse.

These solvers offer two properties that natural-language reasoning lacks: predictable compute scaling and answers that are correct by construction. This motivates an established alternative—the tool-use paradigm [9, 10]—in which, rather than perform every reasoning step internally, a model delegates well-specified tasks to external tools. For instance, arithmetic and general computation can be handled by code interpreters; factual lookup by retrieval systems; and reasoning problems by symbolic solvers. This latter direction is closely related to *autoformalization* [11, 12], which has explored several classes of formal systems and solvers. Logic programming is an especially natural fit for many deductive-reasoning tasks, since it combines declarative representation with automated inference. Yet, compared with code execution and retrieval, logic programming remains comparatively under-served as a standardized tool for language-model agents.

A growing line of work has explored combinations of LLMs and Prolog. [13] use an off-the-shelf LLM to generate Prolog programs and re-try on execution failure; [14] autoformalize game-theoretic scenarios into executable Prolog with an iterative solver-feedback correction loop—a close precedent for the *translate–run–inspect–repair* pattern we propose here. [15] fine-tune LLMs to generate Prolog for arithmetic word problems, while [16] use a Prolog engine to produce verified reasoning trajectories that an LLM imitates as structured chain-of-thought. Closely related, [17] fine-tune a small model with GRPO to use Prolog as a callable tool with internal repair, motivating the primitive that PrologMCP exposes as a standardized interface. Other recent work [18, 19] incorporates structured Prolog representations during training.

However, existing systems are largely bespoke: each defines its own interface, execution model, and error-handling strategy, limiting portability across tasks and agent frameworks. Several open-source Prolog MCP servers also exist (e.g., [20, 21, 22, 23]), exposing query execution and, in some cases, persistent knowledge bases. To our knowledge, however, none provides structured diagnosis of silent failures nor reports an extensive evaluation of a formalizing agent on a reasoning benchmark. This paper presents PrologMCP, a Model Context Protocol (MCP) server that exposes Prolog as a stateful tool supporting a reusable *translate–run–inspect–repair* loop. Our contributions are:

I. **A standardized Prolog tool interface** implemented as an MCP server, providing—beyond query execution—structured proof and failure trees, predicate inspection, hot clause replacement, and session-level isolation that together support agent-driven debugging (Section 3).
II. **A formalizing LLM agent**, translating natural-language problems into Prolog, executing queries using the server, and iteratively refining its representations using structured feedback (Section 4).
III. **An empirical evaluation of the formalizer agent** on `PARARULE-Plus` [24], comparing it against standard LLM and reasoning-model inference (Section 4).

2. Background

Logic Programs in Prolog. Logic programs, most prominently implemented in Prolog [25, 26], provide a declarative representation based on Horn clauses: universally quantified implications of the form "if $b_1, \ldots, b_n$, then h", which map naturally onto facts (clauses with no body), rules (clauses with one or more body literals), and queries (goals to be proved). Reasoning combines *unification* of terms, *SLD-resolution* [27] to reduce a query to subgoals against matching clause heads, and goal-directed search with backtracking. Prolog extends SLD with SLDNF, i.e., negation-as-failure [28], whereby a negated goal is treated as satisfied when the system cannot prove the corresponding positive goal.

Horn clauses align closely with the surface form of natural-language rules ("if X is kind then X is nice" becomes `nice(X) :- kind(X).`), and Prolog's goal-directed inference, support for negation-as-failure, and persistent clause state across calls make it a natural target for autoformalization in

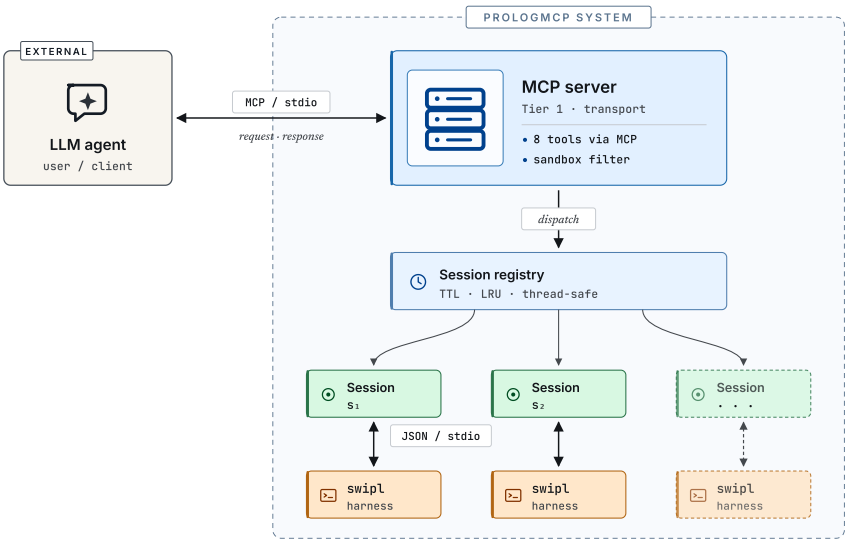


Figure 1: PrologMCP architecture.

deductive-reasoning tasks. Competing approaches such as SAT, SMT, or CP require translation into satisfiability formulas, background theories, or explicit variable domains, and SAT and CP solvers are typically stateless per invocation, which fits multi-turn agent reasoning over a growing knowledge base less well.

Model Context Protocol. The Model Context Protocol [29, 30] provides a standard interface for exposing tools to LLM agents, with conventions for tool discovery, typed argument schemas, and structured error reporting. A tool exposed via MCP can be used by any MCP-aware agent without model-specific glue code. MCP also accommodates stateful tools such as Prolog: across a sequence of calls, an agent can assert clauses, inspect program state, issue queries, receive structured diagnostics, and refine the formalization. The closest precedent is the MCP-Solver discussed in [31], which exposes constraint, SAT, SMT and answer sets backends (MiniZinc, PySAT, Z3 and clingo) but not Prolog-style SLDNF resolution with unification and backtracking over a database of clauses.

Autoformalization. *Autoformalization* is the automatic translation of an informal-language expression into a semantically corresponding expression in a formal language. The term originates in formal mathematics [11], and has since broadened to cover pipelines in which a model emits a formal artefact—a logic program, constraint model, SMT formula, or planning specification—passed to a sound external reasoning procedure. A recent unifying account [12] parameterises autoformalization by an informal language L_i and a formalizable subset thereof, a formal reasoning language L_f with precise semantics, and a semantic equivalence criterion E specifying which aspects of meaning must be preserved. Because E is itself informal and not directly checkable, practical pipelines introduce a computable *validation criterion* V that approximates E, for example by executing the formal artefact with a solver and comparing its output against an expected result.

3. Architecture

PrologMCP is an MCP server that exposes Prolog to LLM agents as a stateful tool. Three properties drive the design: (*i*) *statefulness*—clauses persist across calls, so the agent builds a knowledge base incrementally instead of resubmitting it on every goal; (*ii*) *structured error reporting*—compile- and runtime diagnostics are returned as typed JSON; (*iii*) *isolation*—each session runs in a dedicated interpreter subprocess with bounded resources. Currently, the server supports only SWI-Prolog. Tool signatures and JSON schemas are given in Appendix B. The source code is available on GitHub (https://github.com/dicelab-rhul/PrologMCP).

3.1. Components

The system has three tiers (Figure 1), forming a single path from an agent's tool call down to a Prolog subprocess and back:

1. **A Python MCP server** that presents the tool interface of Section 3.3 to the agent: it declares each tool's input schema, validates and dispatches incoming calls to the appropriate session, and maps harness outcomes—including failures—onto MCP's structured tool-result format. The server holds no Prolog state of its own; it is a thin protocol adapter over the session registry.
2. **A session registry** that owns a pool of Prolog sessions, indexed by a `load_id` returned on session creation. The registry is thread-safe, caps concurrent sessions at a configurable limit, evicts LRU (least-recently-used) sessions on overflow, and reaps sessions idle beyond a TTL (time-to-live) from a background thread.
3. **A Prolog harness** (`harness.pl`), loaded into each session's `swipl` subprocess, that reads newline-delimited JSON commands on `stdin`, dispatches them to handlers, and writes newline-delimited JSON responses on `stdout`. Stderr is reserved for uncaptured diagnostics.

3.2. Session Semantics

A session is created by `consult_text(source, dialect)`: the registry allocates a fresh `load_id`, spawns a `swipl` subprocess with the harness, writes `source` to a file in the session's temporary directory, and issues a `consult` command. If `source` lacks a module declaration, the registry prepends `:- module(m_⟨load_id⟩, []).` so that subsequently defined predicates live in a per-session module disjoint from the harness and from other sessions. Module-qualified goal evaluation (`Module:Goal`) is used throughout, so predicate resolution is unambiguous regardless of the module in which `source` was written.

Syntax errors, singleton warnings, and similar diagnostics are captured by a `message_hook/3` installed in the harness, so a consult that raises errors still loads the clauses it can rather than aborting. Captured diagnostics are returned as a `messages` array, each entry carrying a severity, a kind drawn from a fixed vocabulary, a message string, and, when the diagnostic is attributable to a specific clause, the source line that triggered it. Because the successfully loaded predicates remain callable after a failed consult, the agent can repair the source with `replace_predicate`; consultation is thus *non-fatal*.

The subprocess is held open for the session's lifetime; sessions close explicitly via `close_session`, implicitly on TTL expiry, or by LRU eviction under load.

3.3. Tool Interface

The server exposes eight operational tools plus one administrative tool (Table 1). Each tool corresponds to one primitive the harness supports directly, and the Python layer does no orchestration of its own beyond dispatch and error mapping.

We bound goal evaluation in two ways: by the number of solutions returned per call and by inference depth. A goal that exhausts the depth limit returns with `depth_exceeded: true` and any solutions found so far, rather than raising an exception. Both bounds are surfaced in the response, so the agent can retry with a larger limit when a partial result is inconclusive. These bounds are surfaced in the response so that the agent can retry with a larger limit if the partial result is inconclusive.

4. Evaluation

4.1. Agent implementation

We evaluate three agents that differ in how they produce a final answer: a **Standard** baseline, a **Reasoning** baseline, and a Prolog-augmented **Formalizer**. All agents receive the same natural-language context and yes/no question. **Standard** is a single-call baseline. The model receives the context and question, prefaced by a system prompt that frames the task as logical reasoning under the

Tool	Behaviour
`consult_text`	Open a session from a source string, returning its `load_id`, an optional inventory of defined predicates, and any diagnostics raised during consultation.
`run_goal`	Evaluate a goal string with `findnsols/4` wrapped in `call_with_depth_limit/3`, reporting typed bindings, the number of solutions, and whether the depth bound was hit.
`inspect_predicate`	Report a predicate's clauses, arity, type (`static` | `dynamic` | `foreign`), export status, and clause count; if no exact match exists, suggest predicates with similar names.
`get_source`	Rebuild the module's source by traversing `predicate_property/2` and `clause/2`.
`replace_predicate`	Retract a predicate with `abolish/1`, then reload a single-predicate patch into the session module via `load_files/2`.
`list_messages`	Retrieve the diagnostics gathered over the current session.
`run_tests`	Consult an optional `plunit` file and execute every registered unit, returning pass, fail, and blocked tallies alongside per-test results.
`trace_goal`	Run a depth-bounded meta-interpreter over a goal, yielding a proof tree and its node count.
`close_session`	Tear down the subprocess and remove its temporary directory.

Table 1
Tools exposed over MCP. All arguments and return values are JSON; every error is reported as `{ok: false, error: {kind, message, prolog_term}}` without terminating the session.

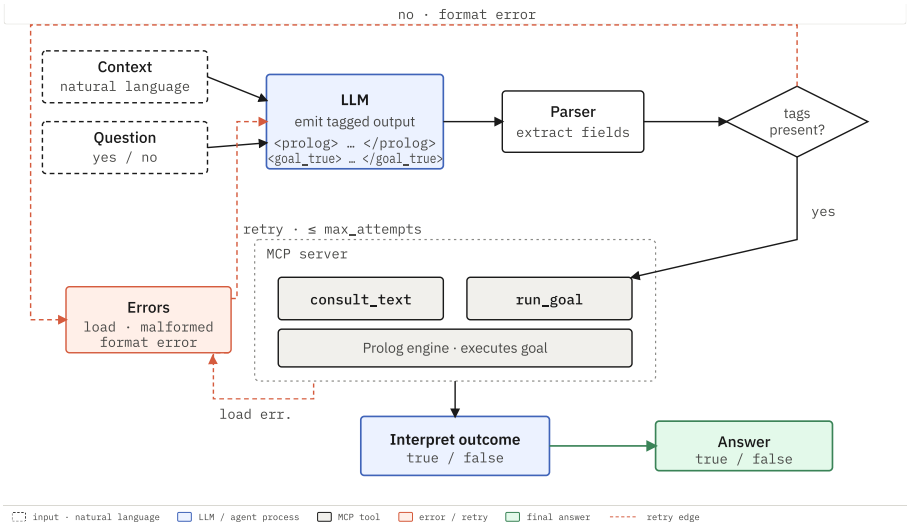


Figure 2: Architecture of the Prolog-augmented Formalizer agent.

PARARULE-Plus negation semantics. It is instructed to answer with exactly one word, `true` or `false`, without elicited intermediate reasoning. **Reasoning** uses the same task formulation as the Standard baseline, but runs it through each provider's reasoning-augmented inference path. For Anthropic models, this corresponds to Extended Thinking mode; for OpenAI models, we use a dedicated reasoning model in place of the standard GPT model.

Formalizer delegates inference to PrologMCP. Given the natural-language context and question, the model generates a Prolog program together with a query goal. The agent loads the generated program through the MCP `consult_text` tool, executes the query through `run_goal`, and interprets the result (Figure 2). To make this interaction robust, the model is instructed to emit the program and goal inside `<prolog>…</prolog>` and `<goal_true>…</goal_true>` tags. A parser extracts these fields; missing or malformed fields are treated as format errors. If parsing fails, if the Prolog program cannot be loaded, or if execution reports an error, the agent returns the structured error

System	Model	Reasoning budget
Standard	`sonnet-4-6`	—
Reasoning	`sonnet-4-6`	10,000 tokens
Formalizer	`sonnet-4-6`	—
Standard	`gpt-4.1`	—
Reasoning	`o4-mini`	medium effort
Formalizer	`gpt-4.1`	—

(a) Per-system model and reasoning-budget settings.

Parameter	Value
Random seed	42
Max formalization attempts	5
Max tokens (Standard, Formalizer)	4,096
Max tokens (Reasoning)	11,024
Temperature (Standard, Formalizer)	0
Temperature (Reasoning)	n/a

(b) Global parameters held constant across all conditions.

Table 2
Experimental configuration.

message to the model and retries, up to `max_attempts` attempts.

The source code for the evaluation framework and experimental logs are available on GitHub (https://github.com/dicelab-rhul/MCPFormalizer). The prompts used in the evaluation are provided in Appendix C: the Formalizer prompt appears in Appendix C.1, and the Standard prompt appears in Appendix C.2.

4.2. Dataset

The `PARARULE-Plus` dataset [24] is a large-scale synthetic dataset for multi-step deductive reasoning over natural language, extending the original PARARULE [32] with deeper inference chains. Each instance is a set of natural-language facts and rules plus a Boolean query (examples in Appendix A). `PARARULE-Plus` covers reasoning depths 2–5 with ≈100 000 samples each, generated over simple people- and animal-attribute domains.

The dataset uses a specific, non-standard interpretation of negation. A body-level `\+ P(x)` succeeds iff `P(x)` is not in the *initial* fact base, regardless of whether it is derivable by rules—resembling negation in semi-positive Datalog [33], where negation is restricted to the extensional database. Queries, by contrast, use the standard closed-world reading over both extensional and intensional facts: a negated query `\+ P(x)` is true iff `P(x)` is neither stated nor derivable. Both standard SLDNF and stratified negation [34] would instead apply the closure uniformly at both levels. This non-standard reading is required to reproduce the ground-truth labels and is passed on to the LLMs in the prompts (Appendix C). This reading is passed to the LLMs by instructing the formalizer agent to record each base fact additionally as `initially(pred, entity)` and to render every body-level negation as `\+ initially(pred, X)` (keeping `initially/2` out of the goal), and by telling the standard and reasoning LLMs to check body-level negations against initial facts only but negated questions against the full closure.

4.3. Experimental setup

We evaluate each agent on two frontier models, Anthropic's Claude Sonnet 4.6 and OpenAI's GPT-4.1, with the Reasoning condition substituting o4-mini in place of GPT-4.1 on the OpenAI side. Per-system parameters are summarised in Table 2a and global parameters in Table 2b. Temperature is fixed at zero for the Standard and Formalizer conditions to make outputs deterministic, but cannot be set uniformly across the evaluation: Anthropic's Extended Thinking and OpenAI's o4-mini both disable temperature control by API constraint, exposing only their respective reasoning-budget parameters in its place. The two Reasoning conditions are therefore compared at the qualitative setting recommended by each provider, namely a 10 000-token thinking budget for Extended Thinking and medium reasoning effort for o4-mini.

The MCP server was started fresh per process via the `prolog-mcp` CLI entry point, and each instance received its own isolated Prolog session, created by `consult_text` and closed on completion. Accuracy is reported as the fraction of instances where the predicted answer matches the ground-truth label, with instances returning `error` or unknown counted as incorrect; no instances returned `error` in any run.

System	Accuracy	Input	Output	Total	Sec.
Claude					
Standard LLM	0.988	**688**	**160**	**849**	**3.57**
Reasoning	**1.000**	717	240	957	4.64
Formalizer (Prolog)	**1.000**	1003	423	1426	5.11
GPT/o4-mini					
Standard LLM	0.762	653	**2**	**655**	**0.98**
Reasoning	0.998	**652**	896	1548	11.27
Formalizer (Prolog)	**1.000**	881	308	1189	4.23

Table 3
Overall results on PARARULE-Plus (400 instances).

We evaluate PrologMCP along two complementary axes. Section 5.1 reports results on a depth-stratified sample of PARARULE-Plus, reflecting the natural distribution of the benchmark across reasoning depths and rule types. Section 5.2 then reports results on a targeted subset that isolates instances where the dataset's negation semantics provably diverges from standard SLDNF evaluation. Together, these evaluations test both the average-case reliability of symbolic delegation and its robustness under adversarial semantic conditions.

5. Results

5.1. Depth-stratified evaluation

Dataset We sampled 400 instances from PARARULE-Plus, stratified by reasoning depth, rule type, and domain. Although PARARULE-Plus differs from SLDNF in its treatment of negation, the sampled subset contains substantial overlap between the two semantics because many body-level negations occur in cases where the distinction is not observable. In 45% of negated predicates, the negated predicate appears only as a base fact and is never derived by any rule. For such predicates, checking absence from the initial facts is equivalent to checking failure of proof, since no rule can derive the predicate. In the remaining 55%, the negated predicate does have derivation rules, so the two semantics can in principle disagree. However, for many specific entities, those rules do not derive the predicate being negated. The proof-theoretic difference is therefore present but not activated by the particular query and knowledge base. Disagreement arises only when the negated predicate is both derivable by rules and actually derived for the entity under consideration; this occurs in 6 instances. Non-termination would also arise when proving the negated predicate requires exploring recursive rule cycles under SLDNF; this would occur in 15 instances.

Main results. Table 3 reports the main results for the three systems described in Section 4.1: the Standard baseline, which answers without an explicit reasoning configuration; the Reasoning condition, which uses a reasoning-enabled inference path or dedicated reasoning model; and the Formalizer, which translates the instance into Prolog and delegates inference to the MCP server.

Overall, both reasoning-augmented approaches improve accuracy over standard LLMs, with the effect most pronounced for the OpenAI models. For Claude Sonnet 4.6, the Standard baseline is already strong, achieving 0.988 accuracy while using the fewest tokens and lowest runtime. Both the Reasoning condition and the Formalizer reach perfect accuracy, but at higher computational cost. One caveat is that, despite the prompt instruction to answer with a single word, Claude Sonnet 4.6 in the Standard condition sometimes produces explanatory reasoning traces. This increases its output-token usage and makes the condition less sharply separated from the explicit reasoning setting.

For the OpenAI models, GPT-4.1 follows the one-word response instruction more closely in the Standard condition, but its accuracy is much lower, at 0.762. Replacing GPT-4.1 with the reasoning model o4-mini raises accuracy to 0.998, while the GPT-4.1 Formalizer reaches perfect accuracy. The

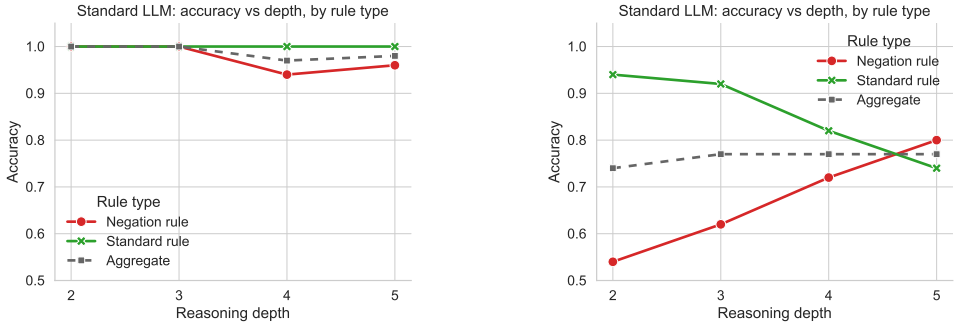
System	$d = 2$	$d = 3$	$d = 4$	$d = 5$
Claude				
Standard LLM	1.000	1.000	0.970	0.980
Reasoning	1.000	1.000	1.000	1.000
Formalizer (Prolog)	1.000	1.000	1.000	1.000
GPT/o4-mini				
Standard LLM	0.740	0.770	0.770	0.770
Reasoning	1.000	1.000	1.000	0.990
Formalizer (Prolog)	1.000	1.000	1.000	1.000

(a) Accuracy by reasoning depth.

System	Negation	Standard
Claude		
Standard LLM	0.975	1.000
Reasoning	1.000	1.000
Formalizer (Prolog)	1.000	1.000
GPT/o4-mini		
Standard LLM	0.670	0.855
Reasoning	0.995	1.000
Formalizer (Prolog)	1.000	1.000

(b) Accuracy by rule type ($n = 200$ per cell).

Table 4
Per-system accuracy broken down by reasoning depth and by rule type.



(a) Claude

(b) GPT/o4-mini

Figure 3: Standard-mode accuracy by reasoning depth and rule type.

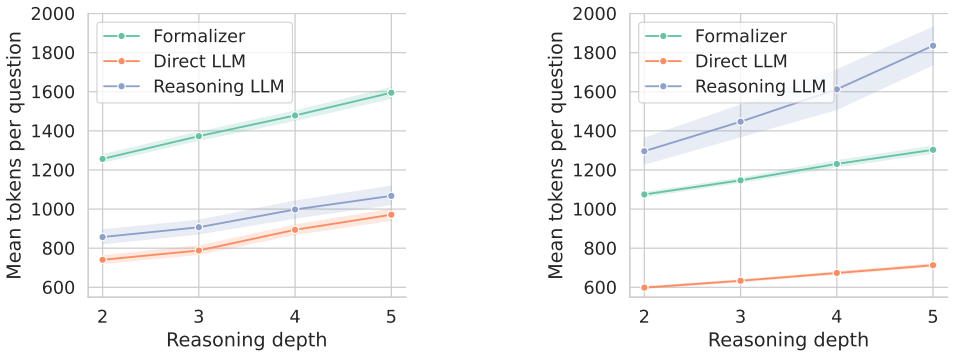
Formalizer is particularly effective in this setting: it achieves the best accuracy with lower total token usage and substantially lower runtime than the reasoning-model condition. These results suggest that symbolic delegation can provide a favourable reliability–cost trade-off when the base model is less reliable, while direct natural-language inference may remain more efficient when the base model is already highly accurate.

Accuracy by depth. The depth-wise results in Table 4a show that standard models become less reliable as reasoning depth increases, whereas both reasoning-augmented approaches remain robust. For Claude Sonnet 4.6, the Standard baseline is perfect at depths 2 and 3, but drops slightly at depths 4 and 5. By contrast, both the Reasoning condition and the Prolog Formalizer achieve perfect accuracy at every depth.

The effect is stronger for the OpenAI models. GPT-4.1 standard prompting remains below 0.80 at every depth, while o4-mini Reasoning reaches perfect accuracy through depth 4 and drops only slightly at depth 5. The GPT-4.1 Formalizer achieves perfect accuracy across all depths, suggesting that explicit symbolic formalization provides the most stable performance as reasoning complexity increases.

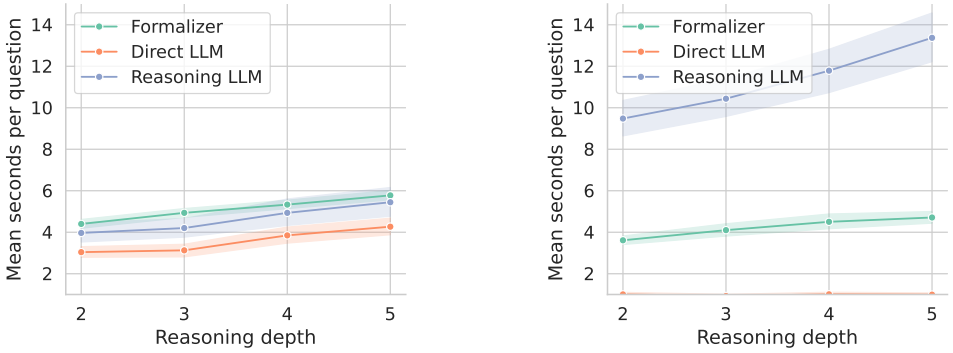
One notable exception to a simple depth-scaling pattern is GPT-4.1 in the Standard condition: its lowest accuracy occurs at depth 1 rather than at the deepest levels. This suggests that the Standard model's errors are not driven solely by reasoning-chain length. Instead, they may reflect reliance on shallow pattern matching, rather than systematic multi-step deduction.

Accuracy by rule type. Table 4b reports accuracy by rule type, and Figure 3 reports accuracy by rule type across reasoning depths. For Claude Sonnet 4.6 in both the Standard and Reasoning conditions, and for o4-mini in the Reasoning condition, all errors occur in instances that contain negation in the rules. For Claude Sonnet 4.6 in the Standard condition, these failures are concentrated at depths 4 and 5



(a) Claude (b) GPT/o4-mini

Figure 4: Token usage by reasoning depth for Claude and GPT models.



(a) Claude (b) GPT

Figure 5: Time elapsed by reasoning depth for Claude and GPT models.

(Figure 3a). This pattern suggests that the non-standard negation semantics of PARARULE-Plus may pose a challenge even for strong reasoning models, because it requires them to suppress more familiar interpretations of negation.

GPT-4.1 in the Standard condition is the only setting in which errors also affect instances without rule-level negation. Figure 3b shows accuracy by depth for both rule types. For GPT-4.1 Standard, accuracy on instances with negation increases with reasoning depth, while accuracy on instances without negation decreases. This non-monotonic pattern is consistent with the hypothesis that GPT-4.1 Standard relies partly on shallow regularities in the input distribution, rather than performing systematic deductive reasoning.

Token usage by depth. Figure 4 shows token usage by reasoning depth. Token consumption increases with depth for all systems, but the magnitude of this growth differs substantially. For Claude Sonnet 4.6, total token usage rises moderately across all methods: Standard increases from 741 tokens at $d = 2$ to 971 at $d = 5$, Reasoning from 857 to 1068, and the Formalizer from 1257 to 1595. For the OpenAI systems, the increase is steepest in the Reasoning condition, where o4-mini grows from 1296 tokens at $d = 2$ to 1835 at $d = 5$. By contrast, the GPT-4.1 Formalizer increases more moderately, from 1075 to 1303 tokens, while GPT-4.1 Standard remains relatively flat, ranging from 598 to 713 tokens across depths.

Overall, reasoning-based approaches, especially o4-mini Reasoning, incur a substantial token overhead as depth increases. The Formalizer uses more tokens than Standard models but its token usage grows predictably with context length and shows the lowest variation across questions. This suggests that symbolic delegation provides a more controlled token-scaling profile than extended natural-language reasoning, while still achieving higher reliability than Standard models.

System	Correct	Incorrect
	Claude	
Standard	843 ± 143 $(n = 395)$	1281 ± 142 $(n = 5)$
Reasoning	957 ± 222 $(n = 400)$	– $(n = 0)$
Formalizer	1426 ± 173 $(n = 400)$	– $(n = 0)$
	GPT/o4-mini	
Standard	656 ± 53 $(n = 305)$	649 ± 48 $(n = 95)$
Reasoning	1547 ± 506 $(n = 399)$	1880 $(n = 1)$
Formalizer	1189 ± 113 $(n = 400)$	– $(n = 0)$

Table 5
Token usage (mean $\pm$ std) by system and evaluation outcome.

Model	System	Total	Incorrect	In SLDNF↓	In loop
Claude	Standard	400	5	3	2
Claude	Reasoning	400	0	0	0
GPT/o4-mini	Standard	400	95	2	1
GPT/o4-mini	Reasoning	400	1	0	1

Table 6
Error breakdown under SLDNF evaluation. SLDNF↓ stands for evaluating to a different label than the gold label under SLDNF.

Runtime by depth. Execution time follows a similar trend to token usage, increasing with reasoning depth for all systems, but with substantial differences in scaling (Figure 5). For Claude Sonnet 4.6, runtime grows gradually across all methods: Standard increases from 3.04 seconds at $d = 2$ to 4.27 seconds at $d = 5$, Reasoning from 3.97 to 5.45 seconds, and the Formalizer from 4.40 to 5.78 seconds. At depths 3–5, the mean runtime of the Reasoning condition approaches that of the Formalizer, suggesting that the additional overhead of symbolic delegation becomes comparatively smaller as reasoning depth increases.

For the OpenAI systems, the contrast is sharper. The o4-mini Reasoning condition rises from 9.48 seconds at $d = 2$ to 13.37 seconds at $d = 5$, reflecting the latency cost of extended reasoning. By contrast, the GPT-4.1 Formalizer increases only from 3.61 to 4.71 seconds, while GPT-4.1 Standard remains nearly constant at around one second across depths. These results indicate that deeper reasoning consistently increases computational cost, but that symbolic delegation has substantially better runtime scaling than the reasoning-model condition, particularly when the non-tool baseline model is less reliable.

Token usage by correctness. Table 5 reports token usage separately for correct and incorrect predictions. Except for GPT-4.1 in the Standard condition, incorrect answers use more tokens on average than correct answers. This pattern is consistent with prior observations that reasoning models often generate more tokens on harder instances [4]. However, the number of failures in the reasoning-augmented conditions is very small.

Causes of errors. Motivated by the observation that most reasoning failures, except for GPT-4.1 in the Standard condition, occur on instances with negation in the rules, we further analysed the evaluation set under standard SLDNF semantics. We identified two classes of instances: (i) instances whose answer under SLDNF differs from the PARARULE-Plus gold label, and (ii) instances for which SLDNF evaluation does not terminate because of recursive rule cycles. Table 6 reports the breakdowns.

For Claude, all five Standard errors fall into one of these two SLDNF-related categories: three correspond to instances where SLDNF gives a different label from the dataset semantics, and two correspond to non-terminating SLDNF evaluations. The single o4-mini Reasoning error also occurs on an instance that loops under SLDNF. By contrast, GPT-4.1 Standard has many errors that are not

Table 7
Results on the 200 SLDNF-diverging instances at reasoning depths 4 and 5. All instances have expected answer *false*. Acc. = accuracy; Tok/q = mean total tokens per question; s/q = mean wall-clock seconds per question.

System	Acc. (all)	Acc. D4	Acc. D5	Tok/q	s/q
Claude					
Reasoning	0.95	0.91	0.99	1345.4	9.5
Formalizer	1.00	1.00	1.00	1548.4	6.2
GPT-4/o4-mini					
Reasoning	0.94	0.94	0.94	2070.0	11.9
Formalizer	0.99	0.98	1.00	1191.7	3.1

explained by these categories, which is consistent with the earlier observation that its failures are not limited to the dataset's non-standard negation semantics. These findings suggest that SLDNF-related semantic mismatch may be the dominant residual failure mode for the strongest systems, but the depth-stratified sample contains too few activating instances (6 diverging answers and 15 non-terminating evaluations) to test this directly. We therefore complement the present evaluation with a targeted set of SLDNF-diverging instances in Section 5.2.

Case study. On instance `NegationRule-D5-28653`, with query `Charlie is smart` and gold answer *true*, the two reasoning models diverge. The intended derivation requires treating `Charlie is dull` as unresolved against the initial facts (it is not stated), so `Charlie is not dull` evaluates to true, after which two rules (*if someone is not dull then they are kind*; *if someone is kind then they are smart*) yield the conclusion. Claude follows exactly this chain. GPT/o4-mini instead pursues `Charlie is dull` as an open subgoal, derives it via forward chaining from other rules, and so concludes the query is false—applying standard SLDNF rather than the prompt's initial-facts reading and constituting a reasoning failure.

Discussion. Across depths and rule types, explicit reasoning and symbolic formalization both substantially improve over standard LLMs, with the largest gains where the base model is weakest: the GPT-4.1 Standard baseline reaches only 0.762, while o4-mini Reasoning and the GPT-4.1 Formalizer reach 0.998 and 1.000 respectively. Most residual errors (except for GPT-4.1 Standard) occur on negation instances and concentrate on cases that either loop or receive a different label under SLDNF, consistent with the case-study failure mode of defaulting to standard negation-as-failure rather than the dataset's initial-facts reading. The depth-stratified sample contains too few such cases to test this directly, motivating the targeted evaluation that follows.

5.2. Targeted evaluation: SLDNF-diverging instances

Dataset. To test the hypothesis that the residual errors observed in Section 5.1 stem from semantic mismatch rather than reasoning depth, we constructed a dedicated set of SLDNF-diverging instances from the negation-type portion of `PARARULE-Plus`. Each instance was converted to SWI-Prolog and evaluated twice: once with the intended `PARARULE-Plus` encoding, in which body-level negation is checked only against the initial fact base, and once with standard SLDNF variant. Instances for which the two evaluations produced different Boolean answers or would loop under SLDNF were labelled *diverging*. We collected 100 such instances at depth 4 and 100 at depth 5, giving a stratified targeted set of 200 instances. All instances in this subset have expected answer *false* under the `PARARULE-Plus` semantics.

Discussion. Table 7 shows that the targeted SLDNF-diverging subset is substantially more challenging for natural-language reasoning than the depth-stratified sample, while the Formalizer remains near-perfect. Claude Reasoning drops from 1.00 to 0.95 and o4-mini Reasoning from 0.998 to 0.94, while the

Formalizer is essentially unchanged (1.00 for Claude, 1.00–0.99 for GPT-4.1). The reliability advantage of symbolic delegation is therefore largest precisely where natural-language reasoning is most fragile.

This robustness is structural: the generated program encodes the PARARULE-Plus semantics explicitly, checking body-level negations against the fixed extensional database rather than against the full derivable closure. The solver is never asked to apply raw SLDNF to those negations, so the main difficulty is shifted from inference to producing a faithful formalization. The depth-wise pattern supports the same interpretation: Claude Reasoning is less accurate at depth 4 than at depth 5, and o4-mini Reasoning is stable across both, so the failures are not explained by reasoning depth but by whether the derivation activates a predicate that is absent from the initial facts yet derivable through rules—the configuration in which PARARULE-Plus and SLDNF disagree.

The cost results reinforce the broader reliability–efficiency trade-off: the Formalizer is faster than the Reasoning condition for both model families (6.2 s vs. 9.5 s for Claude; 3.1 s vs. 11.9 s for the OpenAI systems), and the GPT-4.1 Formalizer is simultaneously more accurate and substantially faster than o4-mini Reasoning while using fewer tokens. Externalising the formal inference step to a Prolog solver is most beneficial when the task departs from interpretations the model would default to, rather than simply requiring longer reasoning chains.

6. Conclusions

We introduced PrologMCP, an MCP-based interface that lets LLM agents delegate symbolic reasoning to a Prolog solver, separating two capabilities often conflated in LLM evaluations: translating an informal problem into a formal representation, and carrying out the resulting inference. The model is responsible for formalization; the solver performs the deductive step.

We evaluated PrologMCP on PARARULE-Plus along two axes: a depth-stratified sample, and a targeted subset that isolates instances where the dataset's negation semantics diverges from standard SLDNF. The Formalizer reached perfect accuracy on the depth-stratified sample for both model families, with the clearest gains over the weaker GPT-4.1 Standard baseline and lower latency than o4-mini Reasoning. On the targeted SLDNF-diverging subset the Formalizer remained near-perfect (1.00 for Claude, 0.99 for GPT-4.1) while the Reasoning conditions dropped to 0.95 and 0.94—exposing a failure mode where models default to standard negation-as-failure even when instructed otherwise. By encoding the intended semantics in the generated Prolog program, the Formalizer sidesteps this and shifts the burden from inference to translation. The approach does not remove the need for accurate semantic translation, motivating future work on validation of generated logic programs and broader benchmark coverage.

7. Limitations and Future Work

The current evaluation relies on a single synthetic benchmark and a closed, frontier-only model set; the cross-system comparison is also not architecturally controlled. PrologMCP itself supports only SWI-Prolog, and its lexical sandbox is not a security boundary. Because all instances formalised on the first attempt, only `consult_text` and `run_goal` were thoroughly exercised. Immediate next steps include broader and harder benchmarks, weaker and open-source models, fine-tuning small models for the formalization step, additional Prolog dialects, container-level isolation, and a routing layer over domain-specialised MCP servers that dispatches sub-problems to the appropriate solver.

Acknowledgments

This work was supported by a Leverhulme Trust International Professorship Grant (LIP-2022-001).

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude Code in order to: Generate code. Further, the author(s) used Claude in order to: Improve writing style. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] J. McCarthy, Programs with Common Sense, Technical Report, USA, 1960.

[2] R. Kowalski, Predicate logic as a programming language, in: IFIP Congress, 1974, pp. 569–574.

[3] A. Newell, H. A. Simon, Computer science as empirical inquiry: Symbols and search, Communications of the ACM 19 (1976) 113–126.

[4] B. Y. Lin, R. Le Bras, K. Richardson, A. Sabharwal, R. Poovendran, P. Clark, Y. Choi, Zebralogic: On the scaling limits of llms for logical reasoning, in: International Conference on Machine Learning, PMLR, 2025, pp. 37889–37905.

[5] M. Kazemi, B. Fatemi, H. Bansal, J. Palowitch, C. Anastasiou, S. V. Mehta, L. K. Jain, V. Aglietti, D. Jindal, Y. P. Chen, et al., Big-bench extra hard, in: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2025, pp. 26473–26501.

[6] N. Dziri, X. Lu, M. Sclar, X. L. Li, L. Jiang, B. Y. Lin, S. Welleck, P. West, C. Bhagavatula, R. Le Bras, et al., Faith and fate: Limits of transformers on compositionality, Advances in neural information processing systems 36 (2023) 70293–70332.

[7] S. I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, M. Farajtabar, GSM-symbolic: Understanding the limitations of mathematical reasoning in large language models, in: The Thirteenth International Conference on Learning Representations, 2025. URL: https://openreview.net/forum?id=AjXkRZIvjB.

[8] P. Shojaee, I. Mirzadeh, M. Horton, S. Bengio, M. Farajtabar, et al., The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, Advances in Neural Information Processing Systems 38 (2026) 108018–108059.

[9] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, T. Scialom, Toolformer: Language models can teach themselves to use tools, Advances in neural information processing systems 36 (2023) 68539–68551.

[10] L. Gao, A. Madaan, S. Zhou, U. Alon, P. Liu, Y. Yang, J. Callan, G. Neubig, Pal: Program-aided language models, in: International conference on machine learning, PMLR, 2023, pp. 10764–10799.

[11] Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, C. Szegedy, Autoformalization with Large Language Models, Advances in Neural Information Processing Systems 35 (2022) 32353–32368. URL: https://proceedings.neurips.cc/paper_files/paper/2022/hash/d0c6bc641a56bebee9d985b937307367-Abstract-Conference.html.

[12] A. Mensfelt, D. Tena Cucala, S. Franco, A. Koutsoukou-Argyraki, V. Trencsenyi, K. Stathis, Towards a common framework for autoformalization, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 40, 2026, pp. 40971–40980. doi:10.1609/aaai.v40i48.42132.

[13] N. Borazjanizadeh, S. T. Piantadosi, Reliable reasoning beyond natural language, arXiv preprint arXiv:2407.11373 (2024).

[14] A. Mensfelt, K. Stathis, V. Tencsenyi, Generative agents for multi-agent autoformalization of interaction scenarios, in: Proceedings of the 28th European Conference on Artificial Intelligence (ECAI 2025), volume 413 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2025, pp. 3759–3766. doi:10.3233/FAIA251256.

[15] X. Yang, B. Chen, Y.-C. Tam, Arithmetic reasoning with llm: Prolog generation & permutation, in: Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers), 2024, pp. 699–710.

[16] X. Tan, Y. Deng, X. Qiu, W. Xu, C. Qu, W. Chu, Y. Xu, Y. Qi, Thought-like-pro: Enhancing reasoning of large language models through self-driven prolog-based chain-of-thought, arXiv preprint arXiv:2407.14562 (2024).
[17] N. Mellgren, P. Schneider-Kamp, L. G. Poech, Training language models to use prolog as a tool, arXiv preprint arXiv:2512.07407 (2025).
[18] P. Zunjare, M. Hsiao, Neuroprolog: Multi-task fine-tuning for neurosymbolic mathematical reasoning via the cocktail effect, arXiv preprint arXiv:2603.02504 (2026).
[19] F. He, Z. Chen, X. Liang, T. Ma, Y. Qiu, S. Wu, J. Yan, Protoreasoning: Prototypes as the foundation for generalizable reasoning in llms, arXiv preprint arXiv:2506.15211 (2025).
[20] adamrybinski, Prolog-MCP Server, https://github.com/adamrybinski/prolog-mcp, 2025. GitHub repository. Accessed 2026-06-19.
[21] umuro, prolog-mcp, https://github.com/umuro/prolog-mcp, 2025. GitHub repository. Accessed 2026-06-19.
[22] rikarazome, prolog-reasoner, https://github.com/rikarazome/prolog-reasoner, 2026. GitHub repository and PyPI package. Accessed 2026-06-19.
[23] vpursuit, MCP Ecosystem, https://github.com/vpursuit/model-context-lab, 2025. GitHub repository. Accessed 2026-06-19.
[24] Q. Bao, A. Y. Peng, T. Hartill, N. Tan, Z. Deng, M. Witbrock, J. Liu, Multi-step deductive reasoning over natural language: An empirical study on out-of-distribution generalisation, arXiv preprint arXiv:2207.14000 (2022).
[25] P. Körner, M. Leuschel, J. Barbosa, V. Santos Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, G. Ciatto, Fifty years of prolog and beyond, Theory and Practice of Logic Programming (2022).
[26] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, Swi-prolog, Theory and Practice of Logic Programming 12 (2012) 67–96.
[27] R. A. Kowalski, D. Kuehner, Linear resolution with selection function, Artificial Intelligence 2 (1971) 227–260.
[28] K. L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Data Bases, Springer, 1978, pp. 293–322.
[29] Anthropic, Introducing the Model Context Protocol, https://www.anthropic.com/news/model-context-protocol, 2024.
[30] X. Hou, Y. Zhao, S. Wang, H. Wang, Model Context Protocol (MCP): Landscape, security threats, and future research directions, ACM Transactions on Software Engineering and Methodology (2026).
[31] S. Szeider, Bridging language models and symbolic solvers via the model context protocol, in: 28th International Conference on Theory and Applications of Satisfiability Testing (SAT 2025), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 30–1.
[32] P. Clark, O. Tafjord, K. Richardson, Transformers as soft reasoners over language, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI-PRICAI 2020), International Joint Conferences on Artificial Intelligence Organization, California, 2020.
[33] S. Abiteboul, R. Hull, V. Vianu, Foundations of databases, volume 8, Addison-Wesley Reading, 1995.
[34] K. R. Apt, H. A. Blair, A. Walker, Towards a theory of declarative knowledge, in: J. Minker (Ed.), Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988, pp. 89–148.

A. PARARULE-Plus

Example instance: non-negation, people

Instance ID: `NonNegationRule-D2-11112`

Context. Charlie is strong. Charlie is high. Charlie is huge. Bob is thin. Bob is small. Erin is quiet. Erin is smart. Erin is kind. Anne is bad. Anne is sad. Anne is rough. Strong people are quiet. If someone is thin and small then they are short. If someone is bad and sad then they are poor. If someone is quiet and smart then they are wealthy. All short people are little. All quiet people are smart. All wealthy people are nice. All poor people are dull. **Query.** Charlie is not smart.

Example instance: negation, people

Instance ID: `NegationRule-D2-6305`

Context. Alan is big. Alan is high. Gary is small. Gary is thin. Fiona is smart. Harry is bad. Harry is poor. If someone is not strong then they are bad. If someone is not sad then they are nice. If someone is smart then they are kind. If someone is kind and not rough then they are quiet. If someone is bad and not strong then they are dull. If someone is small and thin then they are rough. If someone is rough and not kind then they are poor. All nice people are wealthy. **Query.** Fiona is quiet.

Example instance: non-negation, animal

Instance ID: `NonNegationRule-Animal-D2-13824`

Context. The wolf is dull. The wolf is sleepy. The wolf is slow. The wolf sees the mouse. The bald eagle chases the rabbit. The bald eagle is heavy. The bald eagle is big. The mouse is smart. The mouse is kind. The mouse is round. The rabbit is lovely. The rabbit is small. The rabbit is cute. Smart animals are lovely. If something is sleepy then it attacks the mouse. If something attacks the mouse then it is rough. If something is dull and sleepy then it is slow. If something is lovely and small then it is furry. If something is heavy and big then it is awful. All slow animals are lazy. All lovely animals are small. All awful animals are strong. All furry animals are beautiful.

Query. The wolf is not lazy.

Example instance: negation, animal

Instance ID: `NegationRule-Animal-D2-3081`

Context. The bald eagle is sleepy. The bald eagle is rough. The leopard is heavy. The leopard is fierce. The bald eagle visits the rabbit. The leopard sees the dog. The rabbit is nice. The dog is nice. The dog is furry. The dog is lovely. If something is not nice then it needs the rabbit. If something needs the rabbit then it is slow. If something is not round then it is heavy. If something is not strong then it is cute. If something is furry then it is lovely. If something is lovely and not big then it is small. If something is heavy and not round then it is awful. If something is sleepy

and rough then it is big. If something is big and not lovely then it is fierce. All cute animals are beautiful.

Query. The bald eagle is awful.

B. Tool schemas

`consult_text` — *load Prolog source into a new session*

Returns a `load_id` used by all other tools.

```
{
  "type": "object",
  "properties": {
    "source_text":         {"type": "string"},
   "dialect":         {"type": "string", "enum": ["swi"], "default": "swi"},
    "module_name":         {"type": "string"},
    "include_predicates":  {"type": "boolean", "default": false}
  },
  "required": ["source_text"]
}
```

Response keys. `load_id` (string), `module_name` (string), `dialect` (`"swi"`), `messages` (list of `{severity, kind, message, line, column}`), and — when `include_predicates` is `true` — `predicates` (list of predicate descriptors). Compile-time errors are reported through `messages` rather than as a top-level error: the session is created so the agent can inspect and repair.

`run_goal` — *execute a goal in an existing session*

```
{
  "type": "object",
  "properties": {
    "load_id":            {"type": "string"},
    "goal":               {"type": "string"},
    "max_answers":        {"type": "integer", "default": 10},
    "time_limit":         {"type": "number",  "default": 30.0},
    "max_depth":          {"type": "integer", "default": 1000000},
    "allow_side_effects": {"type": "boolean", "default": true}
  },
  "required": ["load_id", "goal"]
}
```

Response keys. `outcome` (`"success"` | `"failure"` | `"timeout"` | `"depth_exceeded"`), `answers` (list of variable-binding maps with typed values), `output` (any captured `stdout` from the goal), `truncated` (boolean: `true` when more answers were available than `max_answers` returned). Goal exceptions are returned as a top-level error with `kind` drawn from `type_error`, `existence_error`, `instantiation_error`, `permission_error`, or `execution_exception`.

`inspect_predicate` — *return the live definition of a named predicate*

```
{
  "type": "object",
  "properties": {
    "load_id": {"type": "string"},
    "name":    {"type": "string"},
    "arity":   {"type": "integer"}
  },
  "required": ["load_id", "name", "arity"]
}
```

Response keys. `found` (boolean), `type` (`static` | `dynamic` | `foreign` | `undefined`), `exported` (boolean), `clause_count` (integer), `clauses` (list of source-form clauses). On miss, `candidates` lists predicates whose name matches up to fuzzy comparison, suitable for repair.

`list_messages` — *return diagnostics accumulated for a session*

```
{
  "type": "object",
  "properties": {
    "load_id":      {"type": "string"},
   "min_severity": {"type": "string", "enum": ["info", "warning", "error"],
                                          "default": "info"}
  },
  "required": ["load_id"]
}
```

Response keys. `messages` (list of `{severity, kind, message, line, column}` records). Messages accumulate across all calls in the session; the caller filters by `min_severity`.

`run_tests` — *run `plunit` tests in the session*

Optionally loads additional test source first.

```
{
  "type": "object",
  "properties": {
    "load_id":          {"type": "string"},
    "test_source_text": {"type": "string"}
  },
  "required": ["load_id"]
}
```

Response keys. `passed`, `failed`, `skipped`, `total` (integers); `results` (list of per-test records with name, status, and failure message if applicable).

`trace_goal` — *trace a goal via a depth-bounded meta-interpreter*

Returns a structured proof tree.

```
{
  "type": "object",
  "properties": {
    "load_id":   {"type": "string"},
    "goal":      {"type": "string"},
    "max_depth": {"type": "integer", "default": 10},
    "max_nodes": {"type": "integer", "default": 200}
  },
  "required": ["load_id", "goal"]
}
```

> **Response keys.** `tree` (nested object with `goal`, `rule`, `children`), `node_count` (integer). Implemented through a custom meta-interpreter rather than SWI-Prolog's tracer; output is structured rather than line-oriented.

`replace_predicate` — *replace the clauses of a single predicate*

`source_text` must define only the target name/arity.

```
{
  "type": "object",
  "properties": {
    "load_id":     {"type": "string"},
    "name":        {"type": "string"},
    "arity":       {"type": "integer"},
    "source_text": {"type": "string"}
  },
  "required": ["load_id", "name", "arity", "source_text"]
}
```

> **Response keys.** `predicate` (echo of `name/arity`), `messages` (list of diagnostics from the patch load, with line and column locators referring to the replacement source). The session is left in its prior state if the patch fails to compile; partial replacement is never observed.

`get_source` — *return the current source of a session*

Reconstructed from all live clauses; reflects any replacements made since load.

```
{
  "type": "object",
  "properties": {
    "load_id": {"type": "string"}
  },
  "required": ["load_id"]
}
```

> **Response keys.** `source` (string).

`close_session` — *terminate a session and release resources*

Releases the subprocess and the session's temporary directory.

```
{
  "type": "object",
  "properties": {
    "load_id": {"type": "string"}
  },
  "required": ["load_id"]
}
```

> **Response keys.** `closed` (boolean). Sessions are also released automatically when the registry's TTL expires or when LRU eviction triggers; explicit close is a courtesy for long-running agents.

C. Experimental prompts

C.1. Formalizer-agent system prompt

Formalizer agent

```
You are a mechanical translator from English to SWI-Prolog.  Your job is \
purely syntactic: map each English sentence to its Prolog equivalent \
one-by-one.  Do not trace, evaluate, or solve the reasoning problem.

Given a CONTEXT (facts and rules) and a yes/no QUESTION, produce:
  1. Valid SWI-Prolog source code that captures every fact and rule.
  2. A single query goal:
       - goal_true  -- SUCCEEDS when the answer is TRUE
       If goal_true fails, the answer is FALSE (closed-world assumption:
       anything not provable from the given facts and rules is false).

SWI-Prolog conventions:
  - All atom names are lowercase and use underscores for spaces:
      `anne`, `is_kind`, `likes_fish`
  - Facts:  `kind(anne).`   `likes(bob, alice).`
  - Rules:  `nice(X) :- kind(X).`
  - Negation-as-failure: `\\+`  (e.g. `not_big(X) :- \\+ big(X).`)
  - Variables start with uppercase: `X`, `Y`, `Who`
  - Do NOT add a module declaration; the server handles namespacing.
  - Do NOT add ``:- discontiguous`` declarations; the server injects them
    automatically for any predicate with multiple clauses.
  - For every predicate that appears ONLY inside ``\\+`` and is not defined
    by any fact or rule, add ``:- dynamic pred/arity.`` -- otherwise
    SWI-Prolog throws existence_error instead of failing cleanly.

Negation -- critical rule (no exceptions, apart from goal):
 Every ``\\+`` in a rule body MUST check ``initially/2``, never the derived
  predicate directly.  Do not reason about whether a predicate could be
  derived -- apply this rule unconditionally to every negation:
  - Assert every base fact in TWO forms: ``pred(entity).`` and
    ``initially(pred, entity).``
  - Write ``\\+ initially(pred, X)`` for EVERY negation in every rule body.

  Example -- "if someone is not high then they are dull":
    dull(X) :- \\+ initially(high, X).   % correct
    dull(X) :- \\+ high(X).              % WRONG -- high/1 may be derived

  Example -- "if someone is sad and not quiet then they are rough":
    rough(X) :- sad(X), \\+ initially(quiet, X).   % correct
   rough(X) :- sad(X), \\+ quiet(X).          % WRONG -- quiet/1 may be derived

Do **not** put ``initially/2`` in the goal.
   Example goal -- "Anne is not sad."
     \\+sad(anne).
```

```
Response format -- output ONLY the XML tags below, no reasoning text:

<prolog>
% facts and rules
fact(entity).  initially(fact, entity).
conclusion(X) :- premise(X).
</prolog>
<goal_true>query_predicate(entity)</goal_true>
```

C.2. Standard-LLM (and Reasoning) system prompt

Standard LLM & reasoning

```
You are a logical reasoning expert.  Given a context (a set of facts and \
rules) and a yes/no question, determine the answer using only the information \
provided.  Use the closed-world assumption: if something cannot be derived \
from the given facts and rules, treat it as false.

Negation has two different readings depending on where it appears.

(1) Negation inside a rule body -- initial-facts reading.
When a rule condition says "not X", check only whether X appears as a \
directly stated fact in the initial context -- not whether X can be derived \
through rules.  Even if a person acquires property X through a chain of \
rules, they still satisfy "not X" in other rule bodies, because X was not \
initially stated for them.
 Example: if "rough" is not stated for Anne but can later be derived, then \
the rule "if someone is not rough then they are smart" still fires for \
Anne -- she counts as "not rough" because rough was not an initial fact.

(2) Negation in the question -- full-derivation reading.
When the question itself asks whether "not X" holds of someone, use \
both facts and rules: "not X(a)" is true \
iff X(a) cannot be derived from the facts and rules (whether directly \
stated or obtained through any chain of rule applications).  Do not apply \
the initial-facts reading here -- it is specific to rule bodies.
  Example: if "energetic" is not stated for Bob but can be derived for \
Bob via some rule chain, then the question "is Bob not energetic?" is \
false, because energetic(Bob) is derivable.  The question "is Bob not \
energetic?" is true only if energetic(Bob) is neither stated nor \
derivable.

Procedure: when evaluating rule bodies, freeze the set of initial facts \
and check negated conditions against that frozen set; when evaluating the \
question, run the full derivation and check negated questions against the \
closure.

Reply with exactly one word: true or false.
  - true   -- the question follows from the context
  - false  -- the question does not follow from the context
```